Aalto University
School of Science
Master's Programme in ICT Innovation

Lukas Brückner

Graphical User Interface Auto-Completion with Element Constraints

Master's Thesis
Espoo, September 25, 2020

Supervisor: Prof. Antti Oulasvirta
Advisors: Dr. Luis Leiva, Aalto University
Jingzhou Du, M.Sc., Huawei Technologies

Aalto University
School of Science
Master's Programme in ICT Innovation

ABSTRACT OF
MASTER'S THESIS

Author: Lukas Brückner

Title:
Graphical User Interface Auto-Completion with Element Constraints

Date: September 25, 2020 **Pages:** vi + 99

Major: Human-Computer Interaction and Design **Code:** SCI3020

Supervisor: Prof. Antti Oulasvirta

Advisors: Dr. Luis Leiva
Jingzhou Du, M.Sc.

Graphical User Interfaces (GUIs) are often the main touchpoint between a digital product and its users. Consistency is a key usability factor of GUIs that dictates reusing layout patterns throughout the designs of a single system. While visual styles can be easily specified across designs, spatial patterns of related elements often emerge in the creation process. Although these patterns are encoded in existing layouts, a large number is difficult to consider manually.

This thesis investigates methods for a design assistance tool that suggests completions for a partial layout. Instead of generating arbitrary completions, we allow the designer to control the next element type and dimensions. The goal of the methods is then to predict the placement of this new element according to the layout patterns of reference designs.

Two recently proposed methods for layout completion were evaluated, a Graph Neural Network (GNN) and a Transformer model, as well as a novel approach that leverages a sequence alignment algorithm and a nearest neighbor search (kNN).

The methods were tested on handcrafted data sets with explicit layout patterns, as well as larger sets of diverse mobile layouts that lack consistent patterns. The implementation of the GNN mostly fails to predict high-quality results. The transformer model captures general layout structures and works reasonably well for spatial compositions that are close to the training data. The kNN approach achieves the best scores overall.

The results suggest that leveraging data sets explicitly via instance-based learning algorithms can outperform neural network approaches for layout design problems. As such, this thesis contributes to establishing smarter design tools for professional designers that increase consistency in the design process.

Keywords: user interfaces, machine learning, layouts, neural networks, design, human-computer interaction

Language: English

Acknowledgments

I wish to thank Prof. Antti Oulasvirta for being supportive, offering different perspectives and being continuously optimistic about my progress; Dr. Luis Leiva for taking the time to regularly discuss my work, offering concrete and helpful suggestions and setting me up on the path of success; Dr. Niraj Dayama for giving me the opportunity to work on this very compelling research problem and being supportive; Du Jingzhou for posing the practical problem, enabling the industrial collaboration with Huawei Technologies, and pointing me to the interesting world of graph neural network research; Simo Santala and Sampo Paukkonen for the great and fun collaboration on creating usable design assistance tools and developing the design software integration; and the rest of the User Interfaces Group for the insightful time into academic HCI-AI research and good conversations.

Espoo, September 25, 2020

Lukas Brückner

Abbreviations and Acronyms

GUI	Graphical User Interface
UI	User Interface, refers to the GUI in this context
UX	User Experience
ML	Machine Learning
GNN	Graph Neural Network
kNN	k-Nearest Neighbor, refers to the Nearest Neighbor method proposed here
NDN	Neural Design Network, the name of the method proposed by [21]
GCN	Graph convolutional network
MLP	Multilayer perceptron
FC	Fully-connected layer

Contents

Chapter 1

Introduction

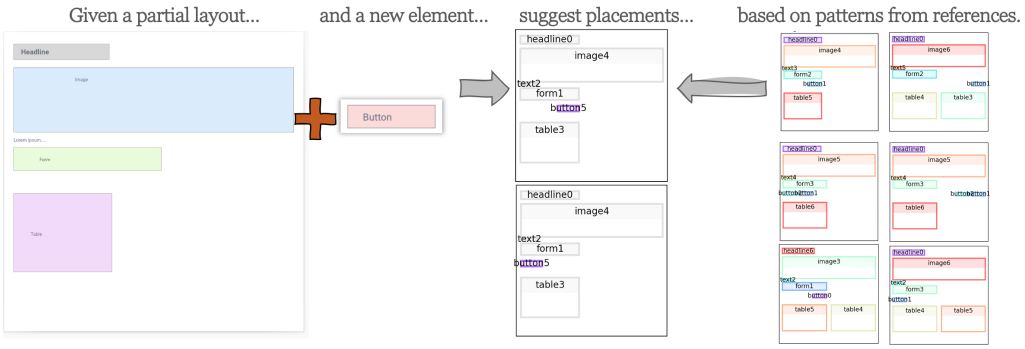

Figure 1.1: Completion of a partial layout with a user-defined element according to layout patterns in reference designs.

Graphical User Interfaces (GUIs) are often the main touchpoint between a digital product and its users. To facilitate the usage of such products and increase user engagement, GUIs are designed to be usable and to provide a good user experience (UX) [37]. One key factor of usability is consistency [34]. In general terms, consistency can refer to any reoccurring feature across designs. This includes visual attributes, such as typography, colors, shapes, etc., but also spatial patterns such as the layout base grid, and recurring patterns of element groups, e.g., forms or navigation bars.

To establish consistency between different screens of a product, companies often create a design system. A design system is typically composed of a large body of rules and guidelines at different design levels, encompassing higher-level layout rules, and lower-level element guidelines. At the highest level, it describes the layouting system which often follows a grid or columns approach. Figure 1.2 shows such a definition that uses columns to align

elements on a screen. Composing a GUI using such a system makes it easy to adapt to different screen sizes, as well as ensuring alignment and visual structure.

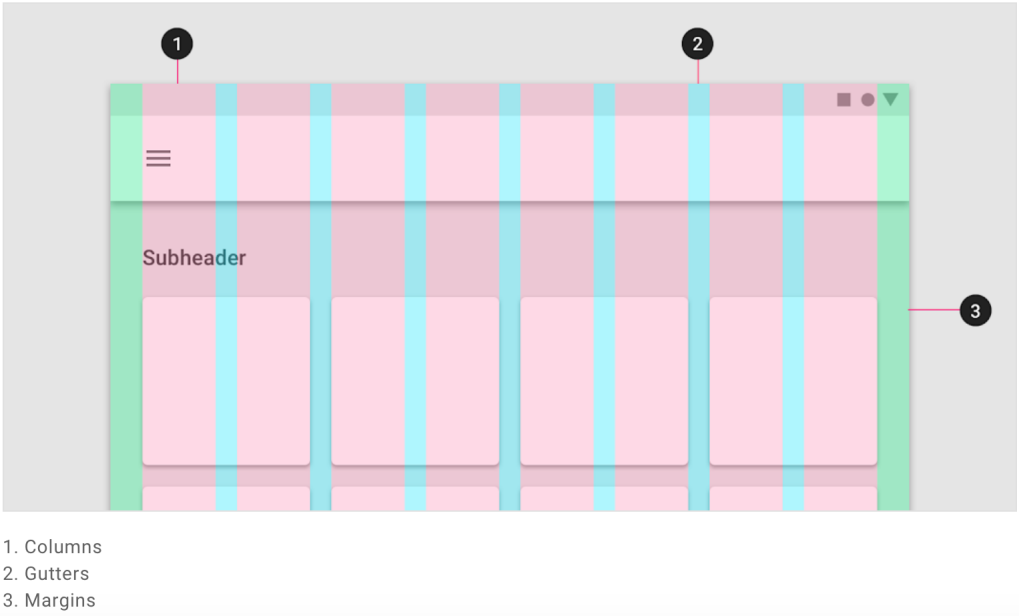

Figure 1.2: The responsive layout system in the Material Design System by Google (source: *Material Design - Google, Inc.* [9]).

At a lower level, the design system defines the specific elements or components of an interface. We understand elements as any item placed on the canvas and refer to defined styles of a comprehensive user interfaces (UI) widget as a component, e.g., a button or input field. Elements are then usually of a specific component type. The component definition might include the dimensions, its internal padding and external margin, as well as its internal composition of lower level atomic elements, such as text elements and shapes and their alignment. Real-world design systems additionally include definitions of the interactions of components, colors, and typography, etc.

These aspects of a design system are defined rigorously to ensure consistency of the overall alignment and the look and feel of components, thus, achieving a high visual consistency. It provides a framework for designers but still leaves enough flexibility and creativity when composing a new layout. Designers need to decide which components and component variations are required to enable users to achieve the goal of the interface, in what order they should be placed and how they interact with each other to provide a satisfying user experience.

Deciding on these compositions of components impacts directly the spatial consistency. Designers should ensure that recurring groups of components are placed similarly, which we call *layout patterns*. These layout patterns might capture how components are placed in relation to other components and are not predefined but emerge as an increasing number of screens are designed. This is depicted in Figure 1.3.

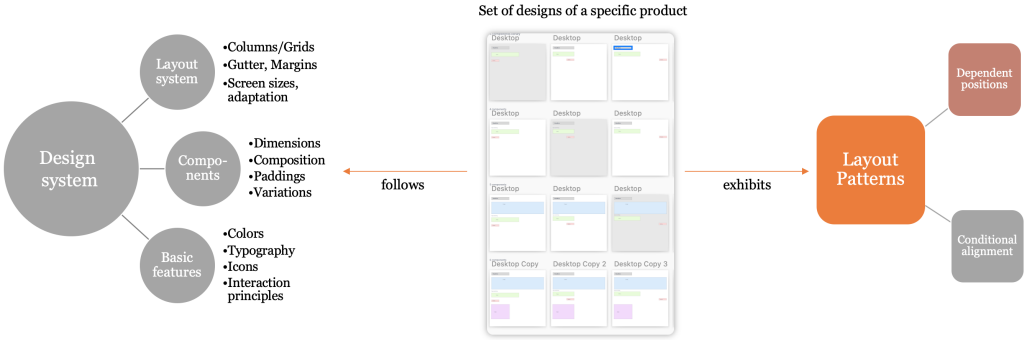

Figure 1.3: Simplified model of how designs of a product follow the explicit rules in a design system while exhibiting layout patterns related to placement and alignment to achieve consistency.

Designers working on a specific product need to take all of these requirements into account which contributes to the substantial amount of time spent on simple and repetitive tasks by designers [35]. To account for this, modern design applications like Sketch, Adobe XD or Figma provide built-in tools that help designers follow general principles of good designs and the basic rules of a design system. It is, for example, possible to define the layout system in terms of columns, gutter, and margin and ensure that elements align to the edges of these columns. Further, designers can define a library of components with specific colors, sizes, and internal compositions which can then be reused across different screens to maintain visual consistency.

But not all requirements of a design system can be defined in the design applications, and designers might not be aware of all layout patterns. This can lead to layouts that are inconsistent with existing designs.

Previously, research has been conducted into representing explicit rules of a design system in assisting tools that aid designers in aligning, packing and optimizing layouts [4, 36]. These works focused on explicit alignment rules of a design system that can be expressed as mathematical requirements and optimized exactly using Integer Programming.

However, assisting designers during the design process to follow layout patterns has not been studied yet to the best of our knowledge. This in turn

can help to create layouts that are consistent and at the same time increase the efficiency of commercial designers.

Since layout patterns are not defined upfront and can emerge during the creation process of UIs for a system, modeling them with explicit methods is not feasible. Instead, we focus on methods of machine learning to understand these patterns from examples.

This thesis investigates solutions that allow suggesting the placement of a new element with a specific type and dimensions onto an existing, partial layout while following layout patterns of reference designs. This is depicted in Figure 1.1. We argue that the human designer should ultimately be in charge of creating UIs. Since a specific task requires specific elements, we give users control over the suggestions and ensure that the user-specified constraints are followed. This is different from previous research in layout completion that involved less control by designers and did not consider layout patterns [11, 21, 25].

1.1 Motivating scenario

The problem was motivated by a scenario given by Huawei Technologies. They own several digital products with teams of user interface designers. As the products became highly comprehensive, the need for ensuring consistent designs became increasingly important, in particular for new designers.

The designers are primarily working with a user interface design tool called *Sketch*[1]. A screenshot of the application is shown in Figure 1.4.

In the application, it is possible to define a set of components with specific styles to be reused across different designs, thus, eliminating the need to manually apply styles to elements when creating a new design. However, with a large number of screens of a product, it becomes difficult to stay aware of all layout patterns and compositions. Their goal is to have an assisting plugin that can recommend the placement of new elements in existing or partial designs.

This use case can be described as follows: Suppose a designer is in the process of creating a new UI design for a new screen of an existing product. They first create the layout architecture that is very close to the other designs of the product that they know. After adding a few of the main elements, they are unsure about how to best place a new button below the form on the page. They remember that there are generally multiple valid locations, however, they want to achieve the highest consistency with other layouts.

[1] https://www.sketch.com

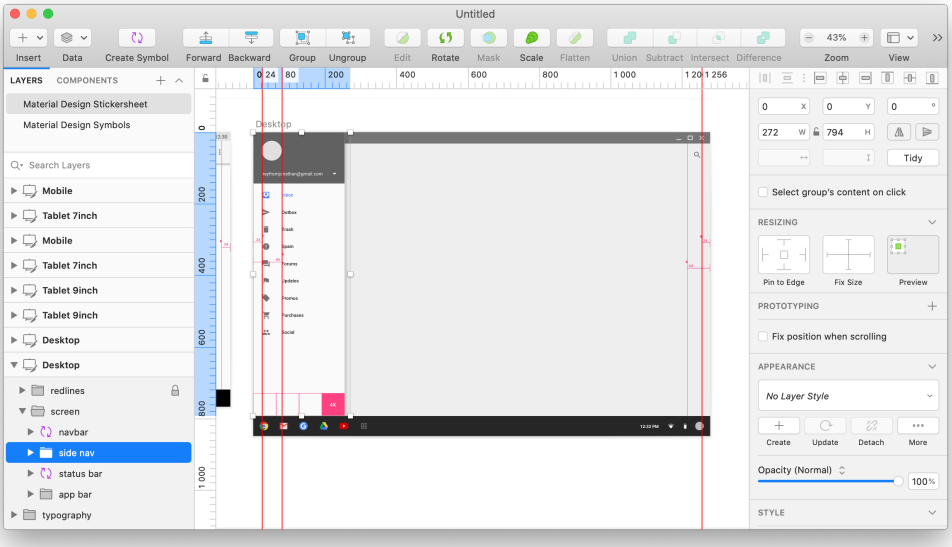

Figure 1.4: Screenshot of the Sketch application. A website is shown in the center and controls to design the UI are placed around it.

In a manual setting, the designer either has to revisit existing designs for similar compositions, which is time-consuming or follow their intuition which might produce inconsistent results. With the help of a design assistant, suggested placements of the button can be given automatically that take into account the layout patterns of previous designs. This helps to ensure consistency and save time.

1.2 Problem statement

This thesis tackles the problem of where to place a new element of a specific type and size onto an existing, partial layout such that the resulting layout is consistent with a set of reference designs.

A consistent placement of a new element is achieved if the layout patterns exhibited in the set of reference designs are followed. We decompose layout patterns into two attributes between elements: (1) Positional dependencies and (2) alignment relations between elements as shown in Figure 1.5. While these attributes do not suffice to describe layout patterns rigorously, we find them to indicate the most important features of such patterns in a similar fashion a human designer would describe it at a high level. For full descriptions, specific margins, offsets, or ranges of valid locations might be needed,

or definitions of edge cases. As we want to create a simple model of layout patterns that can be learned from data, we restrict it in this work to the two attributes mentioned above.

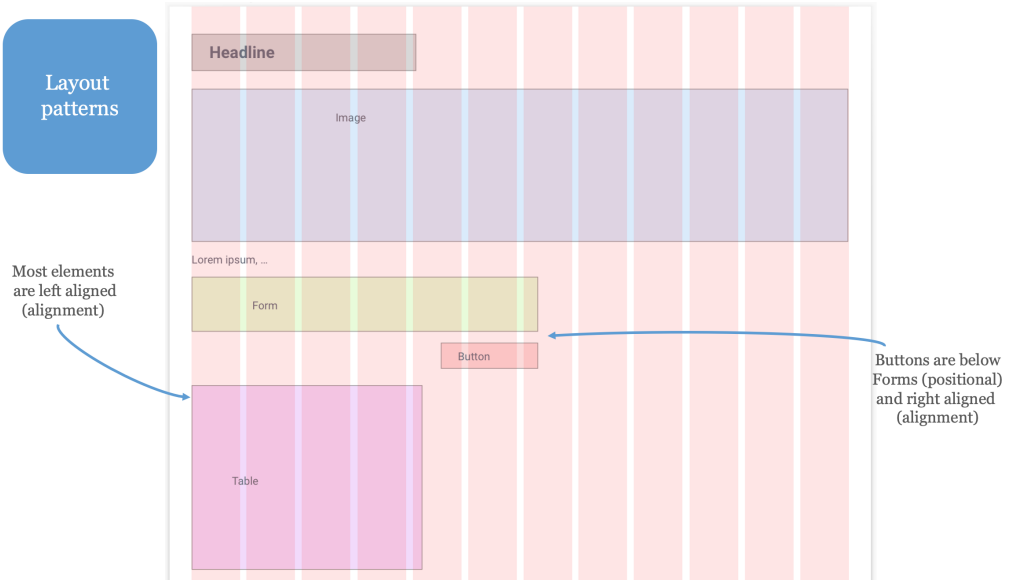

Figure 1.5: A layout exhibiting layout patterns that should be followed when placing new elements in similar compositions.

The attributes can be described in more detail as follows: *Positional* dependency refers to the interplay between elements in which the presence or location of one component influences the relative position of a second component. For example, a button might be placed directly below a form if the form is short while it is placed in a fixed position on the bottom of a screen if the form is long. More specifically, we consider the positional classes *above, below, left, right*. Figure 1.6 shows examples of this relation type. We argue that vertical positioning (*above, below*) is generally more descriptive than horizontal (*left, right*), so, if a component is fully below and to the side of another one, it will be considered as *below* only, and not *left*, or *right*.

On the other hand, *alignment* relations refer to whether elements share common alignment lines of edges or the center, both horizontally and vertically. For example, the button might be always aligned to the left side of a form while an image might be centered on a page. Alignment relations are shown in Figure 1.7. The set of valid relations between two elements depends, however, on their relative positioning. Elements that are *above/below* each other can be either *left, right*, or *vertically center-aligned*. Elements to the *left/right* side of each other can be either *top, bottom*, or *horizontally*

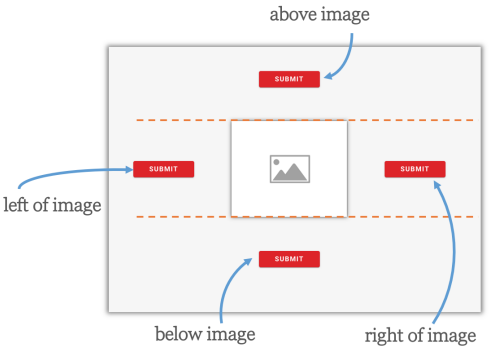

Figure 1.6: Positional relations towards the center image.

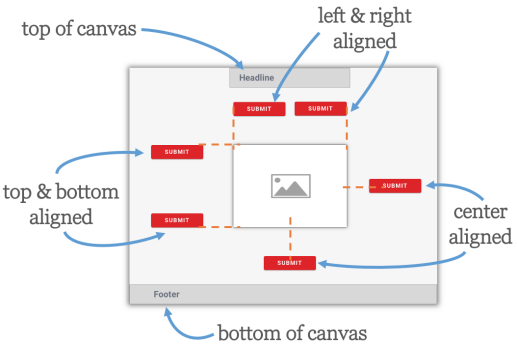

Figure 1.7: Inter-element alignment and relation to the canvas.

center-aligned. Further, we include in this relation category also the relation to the overall canvas of the layout. If an element is within a reasonably small margin to either of the edges of the screen, it will be described as being *at the top, bottom, right* or *left* of the canvas. Finally, as not every element might be aligned in some way to another component, a special *none* alignment type exists that does not restrict the relative placement of the two elements.

In this work, we focus on flat layouts, i.e., those in which there are no elements that are contained in other elements. Instead, every component is assumed to be complete in itself such that it contains every element to make it a meaningful and comprehensive user interface component. This means in turn, that we do not support containers of elements, nor do we expect that any elements overlap, instead, we consider overlap to indicate a bad placement.

We decompose the full design process of a new layout into the sequential process of placing individual elements onto the canvas, one after another. To best assist human designers, the output is expected to provide different variations where possible, and declare a measurement of goodness for these variations. Further, we do not restrict that a group of elements can only have a single pattern placement but there can be multiple valid compositions. As such, we expect the method to return multiple suggestions if possible. Finally, to allow the integration into the design process, the runtime of the method should provide results in a reasonable amount of time that does not impact the flow of the designer (e.g., up to several seconds).

Previous work has shown that objectives of design systems and grid layouts can be achieved via Integer Programming [4, 36]. One of the challenges therein lies in the formulation of every constraint, especially regarding interrelated components. Undocumented patterns that are of interest in our

work, are additionally not known and cannot be expressed a priori for such a combinatorial optimization approach.

1.3 Contribution

This thesis concretizes the general layout completion problem of previous work [11, 25] to apply it to real-world, product-focused design work. We provide a more specific formulation of the problem by focusing on layout patterns as opposed to generic goodness qualities of completions. Further, we contribute by testing different methods on variously sized data sets that should help to understand the potentials and limits of these approaches in realistic, commercial user interface design settings.

Concretely, we evaluate two recently proposed methods for layout completion, a Graph Neural Network (GNN) [21] and a Transformer model [11, 25], as well as our novel approach that leverages a sequence alignment algorithm to calculate features based on layout principles that are used in a nearest neighbor search (kNN).

We test these methods on handcrafted data sets with explicit layout patterns, as well as larger sets of diverse mobile layouts that lack consistent patterns. For the handcrafted data sets, we can explicitly check if layout patterns are followed. For the mobile layouts, we measure general layout qualities to evaluate the results.

Our implementation of the GNN mostly fails to predict high-quality results. The transformer model captures general layout structures and works reasonably well for spatial compositions that are close to the training data. The kNN approach achieves the best overall scores.

Our results suggest that leveraging data sets explicitly via instance-based learning algorithms can outperform neural network approaches for layout design problems. As such, this thesis contributes to establishing smarter design tools for professional designers that increase consistency in the design process.

1.4 Thesis structure

Chapter 1 introduced the motivation for this work and stated the problem of the thesis. In Chapter 2, background information of the problem is explained, as well as basic principles of user interface design. Afterward, Chapter 3 presents related work, ranging from layout generation, layout optimization to layout completion. In Chapter 4, the notation for the layout

methods is introduced, the used layout representations, and the evaluated methods. Then, a short description of the specific implementation follows in Chapter 5. This includes the integration with the design software and details on the machine learning implementation. Afterward, Chapter 6 details how the methods were evaluated. It describes the data sets used for evaluation and the employed metrics. Then, the quantitative results based on prediction accuracy of known patterns, and general layout qualities of the generated layouts are described. Finally, qualitative results for various cases are shown. In Chapter 7, the previously reported results are discussed, including a critical analysis of the limitations. The thesis concludes with Chapter 8 which contains the conclusions to this work and provides an outlook for future work.

Chapter 2

Background

This chapter introduces the reader to the necessary background for understanding the problem of the thesis. First, the details on *Graphical User Interface design* is explained. Then, the theoretical background of the stated problem is given with the *Layout problem*.

2.1 Graphical user interface design

To facilitate the usage of a digital product (such as an application or a website), some kind of user interface is necessary that allows the users of the system to perform tasks with it.

While novel interaction patterns are being researched and are growing, such as voice interfaces, brain interfaces, or virtual reality-based interfaces, the dominant pattern is still the GUI with its direct manipulation interaction technique.

Key elements of such a GUI are components, elements, or widgets (e.g. buttons or form elements) that enable a user to interact via some input device with it and change the state of the application. Ultimately, a system's tasks should be well-supported by its user interface to achieve high usability. One common definition of usability includes five aspects: (1) learnability, (2) efficiency, (3) memorability, (4) safety, and (5) satisfaction [30].

It is then the task of a UI designer to construct a UI design in such a way that it exhibits high usability. The specific way to achieve this depends on the goals of the system, its users, and the context of the usage. Designers, thus, need to consider many factors when building a UI design.

While there is no definite guide to follow in order to achieve usability, different sets of heuristics have been defined that capture common recommendations. A widely adopted list stems from Nielsen developed in 1994 [29]

that lists among other things the heuristic of *consistency and standards*. Consistency allows users to apply learned usage patterns for new user interface areas, improving the efficiency and lowering errors made for completing a task.

As a system becomes more complex, and more designers get involved, achieving this consistency in a single system with multiple user interface windows or screens becomes a challenge. For this reason, many companies introduce *design systems* that include guidelines on the design and layout for the user interfaces of a product as described in the introduction (see Chapter 1).

For a given digital product with a design system, the screen size and the style will be defined and apply across all screens equally. The choice of components and their interaction is a very task- and context-specific problem and must be left to the designer to decide upon. The *layout* aspect of a UI design, however, is a well-understood problem with an active area of research. Layouts are also connected to the notion of consistency, and thus, our aim is to assist in the creation of consistent layouts through computational methods.

2.2 Layout problem

The layout problem is part of a larger family of UI design problems as described by Oulasvirta *et al.* [33]. As such, it has also been formulated for menu assignment, menu ordering, widget selection, etc.

The (2D) layout problem is one where a set of elements need to be placed on a 2D canvas by deciding the pixel coordinates and dimension of every element such that there is no overflow outside the canvas and that other useful layout objectives are fulfilled (see below). One can easily imagine that there are numerous ways of fulfilling these objectives (e.g., without further restrictions elements can be moved pixel-wise, or swapped) as indicated in Figure 2.1, so computer-supported tools are useful to explore and guarantee useful results.

To produce more meaningful and usable layouts more objectives need to be defined. One such problem specification is the *grid layout problem*. As grid or column systems are widely used in website design, it is a logical extension that can also be applied to mobile or desktop user interfaces. With a grid system, all elements in a layout must be aligned to one of the grid lines. This ensures that elements share a common alignment and creates an easier reading flow for the users, especially across multiple screens of a product [1]. More formally, an example of an added objective is to limit or minimize the number of total alignment lines needed in a design, i.e. the more elements

Figure 2.1: The layout problem. Given a set of elements to be placed on a canvas, there is an exhaustive set of possible solutions.

share certain alignment lines, the better.

Concretely, *GRIDS* defined four objectives for well-aligned grid layout generation [4]. *(1) Alignment.* A good layout should have most components aligned with other components to ensure a well-structured and orderly appearance. This is ensured by limiting the number of alignment lines needed in a particular composition. *(2) Rectangularity.* Achieving a non-jagged outline of layouts is an additional objective. *(3) Non-overlapping.* This goal ensures that elements are not in front or behind each other. *(4) Interrelated components.* Components that are related might be required to be placed in closer proximity to each other. Semantic association refers to cases where related elements might need to be placed close to each other because of commonalities (e.g., inputs of the same form should be placed close to each other). Other relative placement requirements can include more general cases when components need to be placed at specific sides of other elements or their presence induces other positional or alignment requirements (e.g., an input label must be always at the top of an input).

When dealing with design systems, the grid may be a pre-defined set of alignment columns at particular locations. In this case, the number of alignment lines is fixed and all elements should be attached to one of these existing alignment lines. In addition, constraints such as specific margins and gutters may be needed to fulfill the requirements of the design system. Finally, also components sizes may be constrained to specific ranges, so it is not possible to resize all components arbitrarily to achieve the other objectives.

This general problem has been approached in different forms by various researchers as we describe in the next chapter.

Chapter 3

Related work

The layout problem or derivations of it have been approached with both combinatorial optimization techniques, machine learning techniques, and others such as Bayesian methods. The following gives a brief overview of the related work in the broad sense. Most work has been conducted in generic *layout generation* or *layout optimization*.

3.1 Layout generation

Hart and Liu first developed a formal description of the layout problem as a rectangular packing problem for integer linear programming in 1995 [12]. This model aimed to fit as many objects as possible onto a window without overlap and clipping of the objects and showed that it can be solved efficiently.

Damera-Venkata *et al.* formulated a probabilistic document model to automatically generate multi-page document compositions given the text and images [3]. It requires a set of probabilistic templates which are evaluated via a Bayesian Network to find the best combination of templates and template parameters to achieve the best document layout. As this application focuses on generating layouts based on templates it is not applicable to our problem where we want to dynamically suggest placements of individual objects based on patterns in the current and the library designs. There might often not be a single reference design that matches the work-in-progress exactly to rely on fixed templates.

Another Bayesian approach was proposed by Talton *et al.* [42]. They learned design patterns from a set of webpages with Bayesian Grammar Induction. The learned grammar can then be used to synthesize new web layouts.

O'Donovan *et al.* developed an energy-based model to automatically

generate graphic designs [31]. Different energy terms are used to model graphic design principles, and the joint model is optimized via simulated annealing. The design principles considered include alignment, symmetric balance, white space, reading flow, overlap, and a saliency model learned via linear regression. Further, it allows learning the model parameters from a small number of example designs via nonlinear inverse optimization such that the style is transferred to the new design. It was then applied to a design assistance tool, but it was simplified to allow real-time feedback while adding user-supplied constraints on the suggestions [32]. In order to facilitate the design process, they showed both refinement suggestions that are close to the edited design, and brainstorming suggestions that show more distant options that could help to inspire the user.

More recently, methods based on generative adversarial networks (GANs) that have shown success in other image generation tasks have been applied to layout generation. With *LayoutGAN*, Li *et al.* proposed a method that captures relations between all elements on a canvas through a stacked self-attention module in the generator based on geometric features and element probabilities [24]. They found that a discriminator that operates in the visual domain and inspects the generated bounding boxes performs better than a discriminator using the generated geometric features directly. It was applied to both document layouts and mobile app layouts but lack control over the generation process.

Zheng *et al.* proposed a content-aware GAN for generating magazine layouts that are conditioned on the desired content, including the topic, keywords, and image contents to be placed [46]. These conditions are embedded into a feature vector that is then concatenated to the latent vector of the generator. It further allows specifying desired locations of certain elements as soft-constraints. To ensure results follow these additional constraints, a large number of results are generated and filtered to contain the desired number of elements per type and rough locations.

Focusing on creative results for graphic designs, Tabata *et al.* created a system that generates layouts based on the user input of the text and images that need to be layouted [41]. It first generates a large number of random layouts with a minimum set of rules such as grid alignment and non-overlapping and then scores candidates based on visual features processed by convolutional neural network layers such that results are similar to real magazine layouts it was trained on. It was further extended to increase the diversity of the results by measuring the cosine similarity of learned features of the candidates [40].

In the field of image scene generation, layout representations of the scenes have been studied, however, their requirements differ quite from UI layouts.

LayoutVAE employs variational autoencoders to predict bounding boxes of target objects for a new scene which is then filled with images to achieve the final scene [17]. They did not model any further constraints of the objects and did not consider alignment as it is not a necessity in natural images. Similarly, Johnson *et al.'s* method of encoding the input as a graph that is then used to generate bounding boxes for a scene image suffers from the same issues while providing a useful mean of defining input constraints that was then expanded in [21].

3.2 Layout optimization and adaptation

Gajos and Weld presented a model to treat user interface adaptation as a discrete constrained optimization problem [6–8]. Their system *SUPPLE++/ ARNAULD* adapts a UI to different device constraints and optimizes it for the user's usage patterns such that the navigation time required to navigate the UI is minimized according to the usage history, and adapts it to a user's motor capabilities to generate a personalized UI.

One early work based on machine learning techniques was *Bricolage* by Kumar *et al.* [18]. It tackled the problem of transferring the content of a web page into the style of another web page. It uses 30 manual features to create mappings between source and target elements and a sophisticated tree-matching algorithm whose weights are learned from human examples via a simple perceptron network. It employed features from the visual domain, like dimensions, font sizes, and colors, as well as structural similarities based on sibling and ancestry nodes.

Leiva developed *ACE* that allows adapting web interfaces to a user's behavior without an explicit model [22]. Instead, it leverages information induced by implicit interaction of the user to inform about the relevance of different parts of the website, which are then slightly modified according to the importance. As such, it uses a straight-forward mathematical formulation for scoring the website elements without models of machine learning or combinatorial optimization or others.

Xu *et al.* proposed an optimization model for alignment improvements in UIs with a sparse linear system solving technique that dynamically evaluates constraints based on the input to find the optimal layout [45]. They noted the issue of resolving ambiguity from the input and studied a gesture-based design tool that allows to interactively update the input constraints to best match the desired properties of the designer.

With *Sketchplore*, Todi *et al.* studied the integration of a layout optimization model into a design sketching tool [43]. It was designed to assist

during the creation process of a new design and inferred the designer's task based on the input and offered optimizations locally, i.e., close to the current design, and globally, i.e., more radical changes. The optimizer used predictive models of sensorimotor performance and perceptions to find better designs. It employed two search heuristics in parallel to explore different solution spaces based on different models, variable neighborhood search, and associative memory. The latter allows using previously created, good designs as starting points for optimization based on similarity to the current design.

Recently, Dayama *et al.* proposed *GRIDS*, a wireframing tool with an integrated grid alignment optimizer based on mixed-integer linear programming [4]. It follows the assumption that many good layouts are following a grid system to define placement areas and optimize results with respect to alignment to the grid, rectangularity of the overall outline, and respecting preferential placement of elements. It followed previous work in design assistance tools to allow ensure diverse results are displayed as opposed to a single optimal design in order to allow designers to explore good options. While it aims to minimize the alignment edges of the elements of the layout, it does not follow a predefined grid system (like a 12 column system) typically found in professional designs and design systems.

This was addressed in the more recent work by Santala [36]. It applied integer programming optimization to design systems and predefined grid systems to ensure designs adhere to the guidelines and rules of such a design system. It was also integrated into a professional UI design tool to assist designers in real-time. As with most explicit methods, it requires to specify all rules beforehand and cannot adapt to conventions easily, as well as not providing a method to decide the placement of new elements.

Further work in this area include *Layout-as-a-Service* [19] and *Scout* [38]. *Layout-as-a-Service* applies layout personalization and optimization to websites with different targets such as selection time, visual saliency, and device size. *Scout* presented a layout exploration tool based on high-level constraints that supports alternative design elements, grouping, placement preferences, and others. As such, it supports exploration for new design ideas but does not consider reference designs.

3.3 Layout completion and assistance

Lee *et al.* proposed the *Neural Design Network* to generate and complete layouts with constraints by representing a layout as a graph and employing graph neural networks [21]. They modeled edges as component relations of relative position and size as input constraints. The input graph is then

completed by connecting all components to each other and predicting labels for these edges. After that, a separate graph network predicts the bounding boxes for each component that is refined in a second step to achieve better alignment. It serves as the basis for our graph network approach that we extend to support better the alignment requirements of design systems.

Nearest neighbor methods have been applied to layout problems as well. *Swire* allows retrieving layouts based on a sketch of a UI by the user [15]. It uses a deep neural network embedding with convolutional layers of the layouts in which neighbors can then be searched for. They showed that it was also effective for retrieving similar layouts based on only a partial layout that then could be used to give inspiration for a designer on how to complete the layout. Naturally, the system only allows to show existing examples and cannot create new variations of combinations that might be needed for more complex UI patterns.

A similar approach was shown with *GUIComp*, a design assistant tool that shows recommendations of the design in the canvas based on a large set of example UIs [20]. The designs are encoded via a stacked autoencoder and a k-nearest neighbor algorithm then retrieves similar designs. As such, it can only give inspiration based on overall similar designs but is not able to show recommendations for a new element to place.

Using transformers, Li *et al.* designed a system that is used to auto-complete a partial mobile app layout [25]. Layouts are represented as sequences based on their tree structure and fed into different transformer structures to predict the types and positions of additional components. Control over the result of the generation was not studied, however.

This was addressed in the more recent work by Gupta *et al.* [11]. They also employed a transformer model to complete a layout based on an input sequence. Instead of encoding an element as a single embedding, they modeled every element as a sequence of attributes and tokens. This allows to condition the generation on partial attributes of the next element. As such, we adapt this proposal in our work and evaluate it according to our problem statement.

Our work builds on the research by Gupta *et al.* [11] and Lee *et al.* [21]. We adjusted the methods to our particular problem statement and addressed the encountered limitations in a novel approach as described in the next chapter.

Chapter 4

Methods

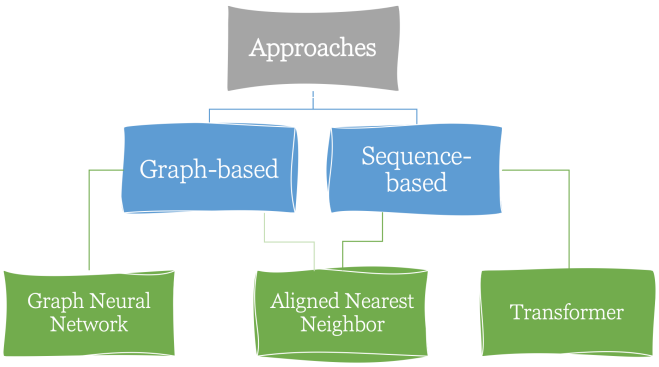

Figure 4.1: We implemented three approaches based on two different layout representations. The graph-based Graph Neural Network, the sequence-based Transformer model and Aligned Nearest Neighbor Search. The latter also takes advantage of the graph representation in the placement stage.

In this chapter, we describe the three methods that are evaluated for the element placement problem, as shown in Figure 4.1. First, we explain the shared *layout notation* for the subsequent *layout representations*. We use two representations based on sequences and on graphs that are employed in the different methods. Then, we describe the *Graph Neural Network* method, followed by the *Transformer* model. The chapter closes with the sequence alignment and *Nearest Neighbor Search* method. The method descriptions go into detail of how the problem is concretely modeled, the procedure of the algorithm, and how a layout with the new element is produced in the end.

4.1 Layout notation

We denote a layout L with a size of $w_L \times h_L$ (width, height) in pixels. It is composed of a set of elements $\{e_i\}$. Every element is defined by $e = (c_e, s_e)$ where $c_e \in C$ is the component (or element type) out of the valid component set C, and $s_e = (x_e^0, y_e^0, w_e, h_e)$ is the 'size box' of the element. x_e^0, y_e^0, w_e, h_e describe the x- and y-coordinates, and the width and height of the element in pixels. This is in contrast to the 'bounding box' $b_e = (x_e^0, y_e^0, x_e^1, y_e^1)$ of an element that describes the upper-left (x_e^0, y_e^0) and lower-right (x_e^1, y_e^1) coordinate points. Finally, when referring to the center points of an element, we use x_e^c, y_e^c for the x- and y-parts accordingly.

For many calculations it is useful to standardize the pixel values between $[0, 1]$, which is achieved by dividing the width and x-coordinates by the layout width w_L, and the height and y-coordinates by the layout height h_L. The corresponding attributes are then denoted with a tilde, $\tilde{x}, \tilde{y}, \tilde{w}, \tilde{h}$. It applies to all variants described above.

In this work, we assume that all elements are rectangular, i.e., that the actual area of an element equals to its bounding box.

Below, we list all layout notations again with a formal description.

L A layout is a composition of a set of elements $\{e_i\}$

w_L, h_L Width and height of a layout L

e Element in a layout that is of a specific component type c_e and described by its 'size box' s_e or 'bounding box' b_e

c A component is a functional user interface element with a specific function, e.g., 'button', 'input field', etc.

C The set of valid components

s The size box of an element (x^0, y^0, w, h)

b The bounding box of an element (x^0, y^0, x^1, y^1)

x^0, y^0 X- and y-coordinates of the upper-left corner of an element

x^1, y^1 X- and y-coordinates of the lower-right corner of an element

w, h Width and height of an element $w = x^1 - x^0$, $h = y^1 - y^0$

x^c, y^c X- and y-coordinates of the centroid of an element

$\tilde{x}, \tilde{y}, \tilde{w}, \tilde{h}$ Standardized measures of coordinates and sizes, as a ratio with the canvas width and height: $\tilde{x} = x/w_L, \tilde{y} = y/h_L, \tilde{w} = w/w_L, \tilde{h} = h/h_L$

$\tilde{\cdot}$ Output of a model is denoted with a hat, especially when denoting predictions

4.2 Layout representations

We employ two complementary layout representations in the evaluated methods: a *sequential* and a *graph-based* representation.

4.2.1 Representing layouts as sequences

Layouts are most naturally depicted on a 2D canvas. However, comparing layouts with different elements at different locations in the layouts is challenging because it is not straight-forward which element from the first layout to compare with from the second layout. A naïve approach would enumerate all possibilities of matching the elements in one layout to the elements in the other layout but this explodes very quickly and becomes intractable. One might try to compare those elements that are closest together but it can fail in cases where similar elements are shifted as additional rows are present in one layout.

It is desirable to simplify the representation such that the possibilities for mappings is greatly reduced and allows more efficient computation. Here, we present our approach of representing layouts as sequences from top-left to bottom-right. In cultures where the reading order of text follows the order from top to bottom and left to right, one can naturally apply this to the reading of visual representations like layout designs. As such, we limit the application of this representation to layouts with left-to-right text.

There are a few commonly referenced layout patterns used in the user interface design field, such as the Z-Pattern, F-Pattern, or Gutenberg diagram, which were shown to be also applicable to web layouts [13]. This enables us to decompose the 2D representation into a simpler flat sequence of elements that allows more efficient comparisons.

Figure 4.2 shows an example of how a layout is represented as a sequence. There might still be ambiguities regarding the natural order in layouts when different hierarchies are present, such as the groups of charts and text. In this work, as we don't consider hierarchies in layouts, there is no clear way of identifying such groups in a general way. Hence, the sequence always follows every virtual row from left to right before adding elements from the row

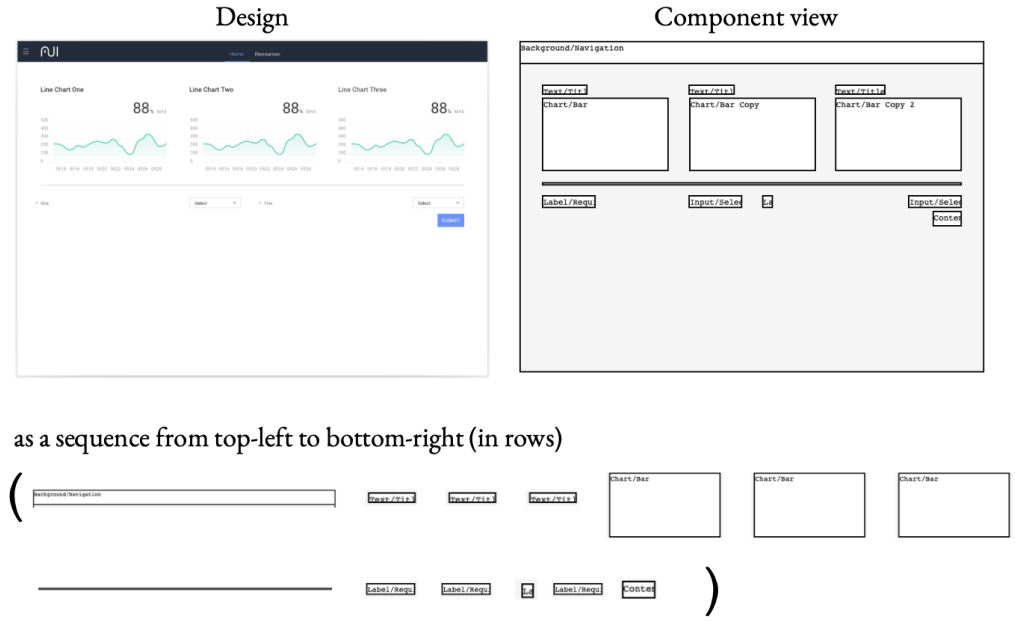

Figure 4.2: Representing a layout as a sequence. The component view shows a simpler bounding box representation. The sequence follows the natural reading order from top-left to bottom-right.

below. Elements that span "multiple rows" are added the first time they are encountered in a row.

More formally, a layout L with n elements $e_i, ..., e_n$ is represented by its sequence S_L as follows:

$$S_L = (e_{p_1}, e_{p_2}, ..., e_{p_n}), \tag{4.1}$$

where e_{p_i} is the ith element according to the placement order p_i.

The placement order p_i is determined according to the reading order from top-left to bottom-right, such that for any two subsequent elements e_{p_i} and $e_{p_{i+1}}$ the following must be true:

$$y^0_{e_{p_i}} \leq y^0_{e_{p_{i+1}}} \wedge x^0_{e_{p_i}} \leq x^0_{e_{p_{i+1}}}, \tag{4.2}$$

where x^0_e, y^0_e represents the x- and y-coordinates of the top-left corner of an element e.

4.2.2 Representing layouts as graphs

A less restrictive representation of a layout can be achieved by a graph. A graph allows capturing the relative positions of its elements more directly

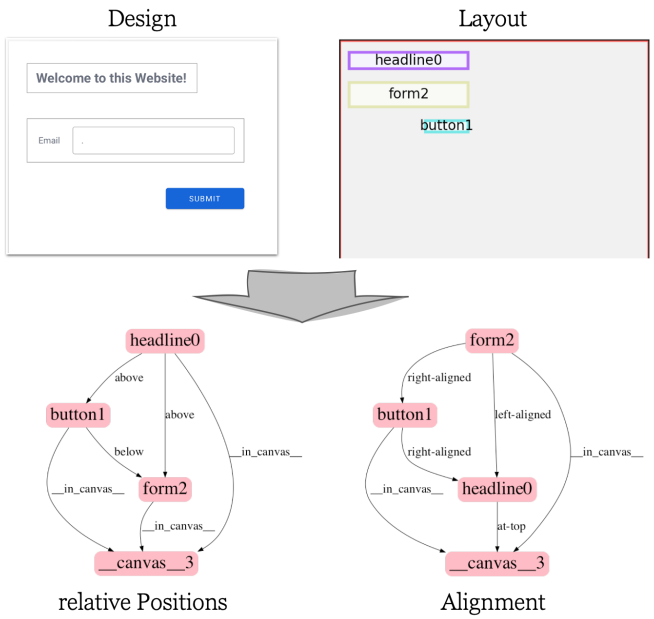

Figure 4.3: Graph representation of a simple layout with 3 elements.

while not operating on an arbitrarily large space with pixel values. In this representation, every element is modeled as a node in the graph, and edges between elements describe some relative features, such as a positional attribute (e.g., 'left', 'above', etc.).

In that sense, the sequence representation from above can be seen as a special case of a graph where every node is solely connected to the elements that come directly before and after it in the top-left to bottom-right reading order.

While one could argue for a sparsely connected graph where an element is connected only to its neighbors, we follow the approach from the 'Neural Design Network' [21] and connect every element to every other element, resulting in a complete graph. This has the advantage that information is able to flow more quickly between all elements, even if information may be redundant.

Since we want to achieve consistent placement of elements according to their positional and alignment relations, two types of edges are added between elements, one for each relation type.

More formally, in this work, layouts are represented as bidirectional heterogeneous graphs $G = (V, A)$, where $V = \{v_i\}$ is the set of vertices (or nodes) that correspond to the elements of a layout $\{v_i\} = \{e_0, ..., e_n\}$, and A is the set of labelled arrows (i.e., directed edges) between vertices (v_i, r, v_j)

where v_i is the source of the edge, v_j its target, and r a relation label from the set R.

To represent both relation types, we use separate graphs G^{pos} and G^{align} where the vertices are the same $V^{\text{pos}} = V^{\text{align}}$ but the arrows A differ in their assigned labels, and based on the valid set R^{pos} and R^{align} respectively.

For every pair of elements e_i, e_j, two directed edges (e_i, r, e_j) and $(e_j, \overline{r}, e_j)$ are created where $\overline{r}$ is the inverse relation of r (e.g., *button –below– text* and *text –above– button*, also see Figure 1.6). This is done for both graphs G^{pos} and G^{align} such that the edges between both are equal except for the relational types. Corresponding to the example above, a pair of alignment relations might be *button –left aligned– text* and *text –left aligned– button*, also see Figure 1.7.

The alignment relation is the same for both directions, while the positional relation is inverted from the different directions. Further, the canvas is represented by a separate node. This canvas node has only incoming edges for specifying the relations of the elements to the canvas but has no outgoing edges. This allows specifying positions to the canvas, e.g., 'at top' or 'at right'. An example layout with a corresponding graph is shown in Figure 4.3.

The number of edges for a layout with this representation is accordingly $2 \cdot 2\frac{n(n-1)}{2} + 2n$ where n is the number of elements. $\frac{n(n-1)}{2}$ is the number of edges in a complete graph which is doubled for the two relation types and the bidirectionality. $2n$ finally is the number of edges going to the canvas object. In the example case with 3 elements, the resulting number of edges is thus 18.

4.3 Graph neural network with constraints

The method follows the description of a 'Neural Design Network' proposed by Lee *et al.* [21]. It consists of a *relation module* that predicts the relation of a new element to the existing elements in the design, followed by a *layout module* that uses the relation graph and the existing canvas to predict a position for the new element, and a *refinement module* that optimizes the result for better layout qualities.

4.3.1 Overview

The network model is depicted in Figure 4.4.

The input query is represented by a directed graph $G = (V, A)$, as explained in the previous section, where vertices $v \in V$ correspond to layout

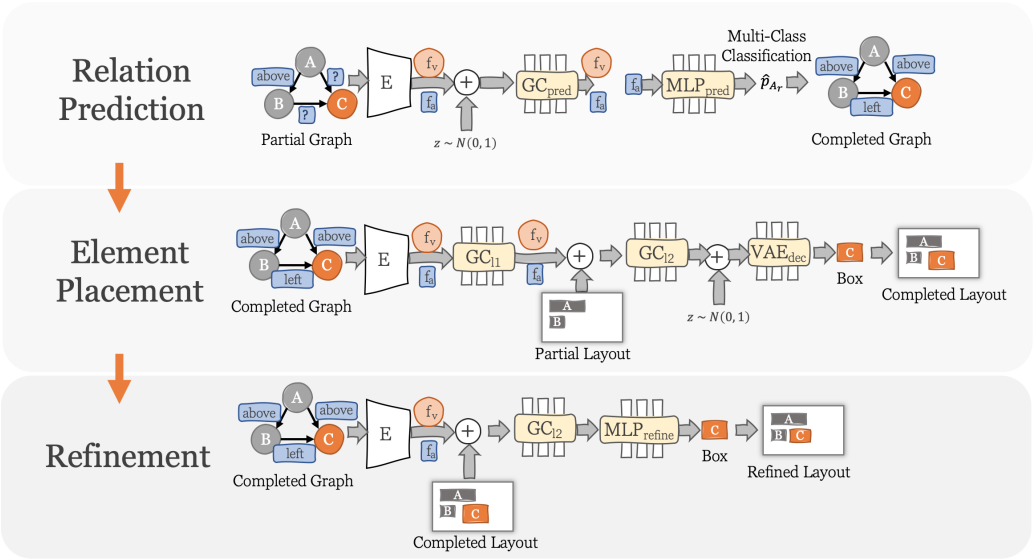

Figure 4.4: The graph neural network implemented according to [21] is composed of three modules to predict a location for a new element.

elements and the arrows $a \in A$ are directed labeled edges describing the relationship between two elements. The vertices always hold information about the component type c. In the layout generation and refinement step, also the existing 'size box' s is part of the vertex data. The canvas is added as a special node with the size box corresponding to the canvas size.

The two relation types (positional and alignment) are encoded in two separate sets of edges A^{pos} and A^{align}. The full graph is, thus, defined by $G = (V, A^{\text{pos}}, A^{\text{align}})$. This is different from the original implementation by Lee *et al.* [21], which used relative *size* as a second edge type. Since in our work the sizes are given by the user, we encode alignment directly instead as it is also part of our target measure. The canvas element only has incoming relations that specify special positions of elements inside the canvas (e.g., 'at top'), if applicable, or simply 'in canvas'.

We use bidirectional edges to enable better information flow between elements because in graph neural networks with directed edges, the information only flows from the source towards the target. Hence, in unidirected graphs, an element might not be informed about other elements if it only contains outward edges.

The new element e_{new} to be added is encoded by a new node in the graph with correct type $c_{e_{\text{new}}}$ and the desired size $(w_{e_{\text{new}}}, h_{e_{\text{new}}})$, such that the size box is $s_{e_{\text{new}}} = (\emptyset, \emptyset, w_{e_{\text{new}}}, h_{e_{\text{new}}})$. Edges between this new element and all other elements are added with a special *unknown* label. Through the

processing modules of the network, a final filled size box for the new element is predicted $\hat{s}_{e_{\text{new}}}$.

In the next sections, we describe the three modules and give details about the graph convolution that is used inside, as well as different variations that were tested.

4.3.2 Relation prediction

The input to the relation module Rel is the partial graph G^{p} where edges to and from the new element c_{new} have the special label *unknown*:

$$\hat{G} = \text{Rel}(G^{\text{p}}). \tag{4.3}$$

This is illustrated in Figure 4.5: adding a new element to a layout means inferring relation types to the existing elements. The goal of this model is to produce labels for these unlabeled edges and thus creating the completed graph $\hat{G}$. In this module, the bounding boxes of the elements are not used as part of the features of the nodes.

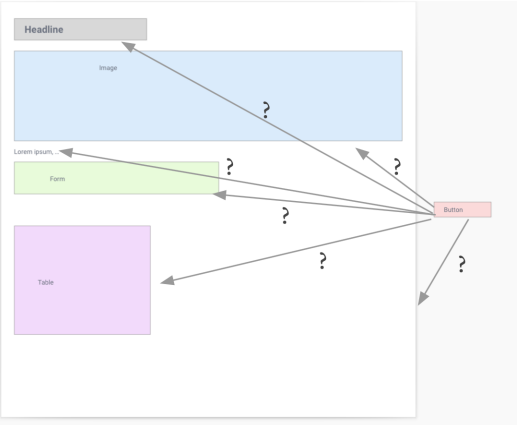

Figure 4.5: Placing a new element on an existing, well-formed layout is formulated as finding relations to the existing elements in the first step.

Depending on the set of training layouts, there might be cases for which the same partial input graph can have multiple different valid output graphs. That is why we condition the graph completion task on a learned latent variable z_{rel} that allows differentiating between multiple variations of the same input.

First, the categorical node and edge labels are turned into embeddings, in the same way, words are embedded in natural language processing tasks:

$$F_{V^{\mathrm{P}}} = \mathrm{Embed}_{|C| \times D_{\mathrm{embed}}}(V^{\mathrm{P}}), \tag{4.4}$$

$$F_{A^{\mathrm{P}}} = \mathrm{Embed}_{|R| \times D_{\mathrm{embed}}}(A^{\mathrm{P}}), \tag{4.5}$$

where $V^{\mathrm{P}}, A^{\mathrm{P}}$ represent the nodes and edges of the partial graph, Embed is an embedding function, $|C|, |R|$ correspond to the vocabulary size of the embedding functions, component types and relation types respectively, and D_{embed} is the embedding dimension. The output $F_{V^{\mathrm{P}}}, F_{A^{\mathrm{P}}}$ are the feature matrices for all vertices and edges. It is a matrix of size $|C| \times D_{\mathrm{embed}}$ respective $|R| \times D_{\mathrm{embed}}$. The embedding dimension has to be the same size for both nodes and edges for usage in the graph convolution layers. The individual feature vector of nodes and edges are given as $f_v \in F_{V^{\mathrm{P}}}$ and $f_a \in F_{A^{\mathrm{P}}}$ respectively.

Such embeddings learn a multi-dimensional dense vector of real numbers for each category that should encode semantic similarities between categories or other properties that are relevant for the later layers of the network and the task of the model. These embeddings are not predefined but are part of the parameters of the deep learning network and are learned during training so it can learn to produce the semantically similar embeddings for semantically similar categories.

Next, a latent vector of a specific dimension $D_{z_{\mathrm{rel}}}$ is either sampled from a standard normal distribution $z_{\mathrm{rel}} \sim \mathcal{N}(0, 1)$ (during inference), or encoded from the ground truth graph $z_{\mathrm{rel}} = \mathrm{Enc}(G^*)$ as described in paragraph 4.3.2 below during training to allow reconstruction of different ground truth graphs G^* from the same input.

The latent vector is concatenated onto the feature vectors of the nodes and edges. The resulting feature vectors are, thus, updated to:

$$F'_V = (F_V, z_{\mathrm{rel}}), \tag{4.6}$$

$$F'_A = (F_A, z_{\mathrm{rel}}), \tag{4.7}$$

and the dimension per node and edge is now $|f_v| = |f_a| = D_{\mathrm{embed}} + D_{z_{\mathrm{rel}}}$.

These feature vectors F'_V, F'_A are the inputs to the actual graph convolution network $\mathrm{GCN}_{\mathrm{pred}}$ which is described in subsection 4.3.5. It produces updated feature vectors for both nodes and edges:

$$F''_V, F''_A = \mathrm{GCN}_{\mathrm{pred}}(F'_V, F'_A). \tag{4.8}$$

Finally, the convoluted feature vectors of the edges F''_A are fed into a multilayer perceptron network $\mathrm{MLP}_{\mathrm{pred}}$ that performs a multi-class classification task on each input vector, and returns a probability vector on the relation categories for each edge:

$$\hat{p}_{A_r} = \sigma(\mathrm{MLP}_{\mathrm{pred}}(F''_A)), \tag{4.9}$$

$$\{\hat{r}_i\} = \arg\max(\hat{p}_{r_i} \in \hat{p}_{A_r}) \quad \forall i \in [0, |A|). \tag{4.10}$$

$\hat{p}_{r_i} \in \hat{p}_{A_r}$ are the probability vectors for the label of every edge as returned by a softmax activation function σ. The output is then an $|R|$-dimensional vector that assigns a specific score to each possible label category for every input edge. The predicted label is then returned by computing the arg max on this vector.

The missing edge labels are learned and predicted independently for the two relation categories. Hence, we create actually two models $\mathrm{Rel}^{\mathrm{pos}}, \mathrm{Rel}^{\mathrm{align}}$, one for each relation type.

By replacing the unknown edge labels from the partial graph with the predicted labels from the network, a complete graph $\hat{G}$ is generated that can then be used in a subsequent layouting module.

Latent vector encoding The latent vector of dimension $D_{z_{\mathrm{rel}}}$ is encoded from the ground truth graph G^* as described here:

$$z_{\mathrm{rel}} = \mathrm{Enc}(G^*), \tag{4.11}$$

where $G^* = (V^*, A^*)$ is a ground truth graph during training.

$\mathrm{Enc}(G^*)$ is composed of the following steps. First, the nodes and edges are embedded, similar to the description before:

$$F_{V^*} = \mathrm{Embed}^{\mathrm{enc}}_{|C| \times D^{\mathrm{enc}}_{\mathrm{embed}}}(V^*), \tag{4.12}$$

$$F_{A^*} = \mathrm{Embed}^{\mathrm{enc}}_{|R| \times D^{\mathrm{enc}}_{\mathrm{embed}}}(A^*). \tag{4.13}$$

The embedding functions and dimensions are different from the previous step, as denoted with the superscript $^{\mathrm{enc}}$.

Next, the graph features are convoluted in a graph convolution network:

$$F'_{V^*}, F'_{A^*} = \mathrm{GCN}_{\mathrm{enc}}(F_{V^*}, F_{A^*}), \tag{4.14}$$

where $\mathrm{GCN}_{\mathrm{enc}}$ is a graph convolution network, and F'_{V^*}, F'_{A^*} are the updated features of the vertices and edges.

These vectors are then passed through a two-headed dense neural network $\mathrm{MLP}_{\mathrm{enc}}$, to generate μ, σ:

$$\mu, \sigma = \mathrm{MLP}_{\mathrm{enc}}(F'_{V^*}, F'_{A^*}). \tag{4.15}$$

Finally, via reparameterization, the latent vector is generated:

$$z_{\text{enc}} = \mu + \sigma\epsilon, \tag{4.16}$$

where ϵ is a random noise variable sampled from $\mathcal{N}(0, 1)$ and z_{enc} has the dimension $D_{z_{\text{rel}}}$.

Loss The network is trained with a reconstruction loss $\mathcal{L}_{\text{rec}}$ on the category prediction, and an entropy loss $\mathcal{L}_{\text{kl}_1}$ between the generated latent vectors z_{rel} and the prior distribution $\mathcal{N}(0, 1)$:

$$\mathcal{L}_{\text{rec}_1} = \text{CE}(\{\hat{r_i}\}, \{r^*\}, w), \tag{4.17}$$

where $\text{CE} = -\sum_n^{|R|} r_n^* \log \hat{r_n}$ is the cross-entropy function, $\{\hat{r_i}\}$ the set of predicted relation categories, and $\{r^*\}$ the set of true relation categories. w is a weight parameter to account for imbalanced data sets, especially with smaller ones, which is automatically calculated according to the data imbalance, and

$$\mathcal{L}_{\text{kl}_1} = \text{KL}(z_{\text{rel}}, \mathcal{N}(0, 1)), \tag{4.18}$$

where KL is the Kullback-Leiber divergence function, z_{rel} the encoded latent variable, and $\mathcal{N}(0, 1)$ the standard normal distribution.

Both losses are summed with weights to form the complete loss for the relation module:

$$\mathcal{L}_{\text{rel}} = \lambda_{\text{rec}_1}\mathcal{L}_{\text{rec}_1} + \lambda_{\text{kl}_1}\mathcal{L}_{\text{kl}_1}. \tag{4.19}$$

4.3.3 Layout generation

In the layout generation module, the completed graph $\hat{G} = (V, \hat{A})$ along with the elements size boxes $\{s_{e_i}\}$ is used to predict the size box of the new element $\hat{s}_{e_{\text{new}}}$ such that a complete layout is created:

$$\hat{s}_{e_{\text{new}}} = \text{Layout}(\hat{G}, \{s_{e_i}\}). \tag{4.20}$$

We do not need to work with the two relation types separately in the layout module, so the edges are merged and the vocabulary of the relation types is composed of the joint vocabularies of the different types.

As before, the first step is creating embeddings for the nodes and edges:

$$F_V = \text{Embed}_{|C| \times D_{\text{embed}}}(V), \tag{4.21}$$

$$F_{\hat{A}} = \text{Embed}_{|R| \times D_{\text{embed}}}(\hat{A}). \tag{4.22}$$

A graph representation is generated by passing these features through a graph convolution network:

$$F'_V, F'_{\hat{A}} = \mathrm{GCN}_{l1}(F_V, F_{\hat{A}}). \tag{4.23}$$

In the original description of the method, an iterative process is used to generate the boxes for new elements, as they were supporting multiple new elements, or empty layouts as well. To be consistent with the stated problem and the other methods, we only consider a single element to be added and, thus, employ a single prediction step for a given input layout.

To generate a placement for a new element, the existing standardized size boxes $\{\tilde{s}_{e_i}\}$ are concatenated to the feature vectors of the nodes. To the new element e_{new}, the half-empty size box is concatenated $\tilde{s}_{e_{\mathrm{new}}} = (\emptyset, \emptyset, \tilde{w}_{e_{\mathrm{new}}}, \tilde{h}_{e_{\mathrm{new}}})$:

$$F''_V = (F'_V, \{\tilde{s}_{e_i}\}). \tag{4.24}$$

To adjust the dimensionality of the edge vectors F_A to match the node vectors, a 4-dimensional vector of zeros is concatenated onto them:

$$F''_{\hat{A}} = (F'_{\hat{A}}, (0, 0, 0, 0)). \tag{4.25}$$

These features are passed through another graph convolution network GCN_{l1}, and a variational auto encoder-decoder network of fully-connected layers $\mathrm{h}_s^{\mathrm{enc}}, \mathrm{h}_s^{\mathrm{dec}}$ generates the box prediction from the new node feature of the new element e_{new}:

$$F'''_V, F'''_{\hat{A}} = GConv_{l2}(F''_V, F''_{\hat{A}}), \tag{4.26}$$

$$z_{\mathrm{lay}} = \mathrm{h}_s^{\mathrm{enc}}(s^*_{e_{\mathrm{new}}}), \tag{4.27}$$

$$\hat{s}_{e_{\mathrm{new}}} = \mathrm{h}_s^{dec}(F'''_{e_{\mathrm{new}}}, z_{\mathrm{lay}}), \tag{4.28}$$

where $s^*_{e_{\mathrm{new}}}$ is the ground truth box of the new element during training, during which the latent code is generated through the encoder network from the ground truth box. During inference, the latent code z_{lay} is sampled from a prior distribution.

Loss The network is trained with a reconstruction loss $\mathcal{L}_{\mathrm{rec}_2}$ on the size box differences, and an entropy loss $\mathcal{L}_{\mathrm{kl}_2}$ between the generated latent vectors z_{lay} and the prior distribution $\mathcal{N}(0, 1)$. Additionally, to prioritize the given sizes of the elements, another loss $\mathcal{L}_{\mathrm{rec}_3}$ is added:

$$\mathcal{L}_{\mathrm{rec}_2} = ||\hat{s}_i - s^*_i||_1, \tag{4.29}$$

where $||\cdot||_1$ is the L1-loss between the predicted box and the ground truth box s^*,

$$\mathcal{L}_{\text{rec}_3} = ||(\hat{w}, \hat{h}) - (w^*, h^*)||_1, \tag{4.30}$$

where w, h represent the widths and heights of new elements, and

$$\mathcal{L}_{\text{kl}_2} = \text{KL}(z_{\text{lay}}, \mathcal{N}(0, 1)), \tag{4.31}$$

where KL is the Kullback-Leiber divergence function, z_{lay} the encoded latent variable, and $\mathcal{N}(0, 1)$ the standard normal distribution.

As before, the losses are summed with weights to form the complete layout loss:

$$\mathcal{L}_{\text{lay}} = \lambda_{\text{rec}_2}\mathcal{L}_{\text{rec}_2} + \lambda_{\text{rec}_3}\mathcal{L}_{\text{rec}_3} + \lambda_{\text{kl}_2}\mathcal{L}_{\text{kl}_2}. \tag{4.32}$$

4.3.4 Refinement module

The final refinement module operates on the generated complete layout with the goal of fine-tuning the previous result and producing a more aesthetically pleasing layout.

It is input the completed graph $\hat{G} = (V, \hat{A})$, along with the size boxes of the existing layout $\{s_{e_i}\}$ and the size box of the new element for a query $\hat{s}_{e_{\text{new}}}$:

$$\hat{s}'_{e_{\text{new}}} = \text{Refine}(\hat{G}, \{s_{e_i}\}, \hat{s}_{e_{\text{new}}}). \tag{4.33}$$

As before, the graph $\hat{G}$ is first embedded, producing $F_V, F_{\hat{A}}$. Next, the completed layout from the previous model is concatenated onto the features of the node features: $F'_V = (F_V, \{s_{e_i}\})$. In this module, we can directly adjust the dimensions of the embedding functions to produce differently sized embeddings for the nodes and edges, such that the feature dimensions match after concatenating the size boxes: $D^{\text{refine}}_{\text{embed}_V} = D^{\text{refine}}_{\text{embed}_A} - 4$.

The final features are then convoluted via a graph convolution network $\text{GCN}_{\text{refine}}$, and the resulting feature vector of the new element is fed into a multi-layer perceptron network $\text{MLP}_{\text{refine}}$ to generate the refined size box $\hat{s}'_{e_{\text{new}}}$:

$$F''_V, F''_{\hat{A}} = \text{GCN}_{\text{refine}}(F'_V, F_{\hat{A}}), \tag{4.34}$$

$$\hat{s}'_{e_{\text{new}}} = \text{MLP}_{\text{refine}}(F''_{e_{\text{new}}}). \tag{4.35}$$

Training and loss To teach the network to fix layouts, the training input layouts are used and a random perturbation of $\delta = U(-0.05, 0.05)$ is applied to the coordinates of new elements $s^*_{e_{\text{new}}} = (\tilde{x}_{e_{\text{new}}} + \delta, \tilde{y}_{e_{\text{new}}} + \delta, \tilde{w}_{e_{\text{new}}}, \tilde{h}_{e_{\text{new}}})$.

The boxes of the other elements are kept as is such that there are well-aligned elements to align to.

The network is trained with a reconstruction loss $\mathcal{L}_{\text{rec}_3}$ on the size box differences:

$$\mathcal{L}_{\text{rec}_3} = ||\hat{s}'_i - s^*_i||_1, \tag{4.36}$$

where $||\cdot||_1$ is the L1-loss between the predicted box and the ground truth box s^*.

4.3.5 Graph convolution network

To perform the actual graph convolutions, we use the same approach as described in the original 'Neural Design Network' [21] which in turn uses the method as described in 'sg2im' by Johnson *et al.* [16].

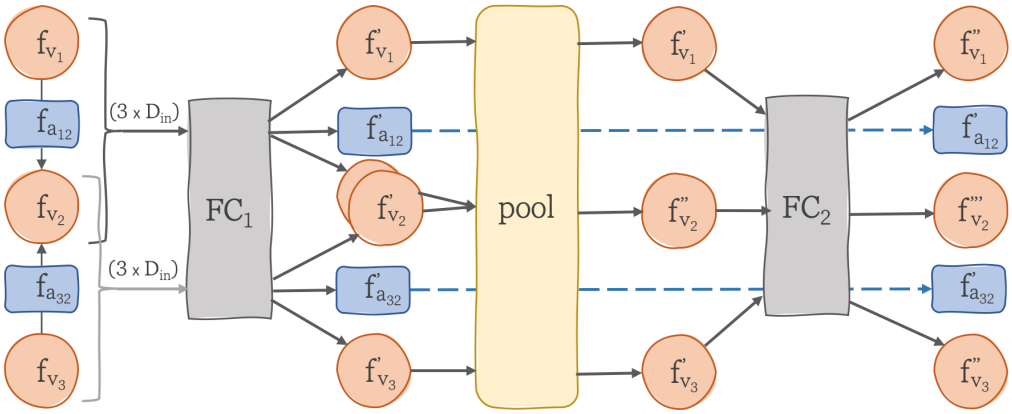

Figure 4.6: The graph convolutional layer. f_v represents the features of a vertex, f_a the features of an arrow (edge). FC describe fully-connected layers.

This graph convolution layer GC is depicted in Figure 4.6. The input are feature vectors of the vertices f_{v_i} and of the arrows f_{a_i} with dimension D_{in}. First, they are passed as triples in the form of $(f_{v_i}, f_{a_i}, f_{v_j})$ where v_i is a source of the triple a_i and v_j the target through a fully-connected neural network FC_1, resulting in $(f'_{v_i}, f'_{a_i}, f'_{v_j})$.

Vertices that appear multiples times in any of the triples are then pooled in the next step which is by default implemented as an *average* pooling, creating $\{f''_{v_i}\}$. The pooled vertex features are finally passed through a second fully-connected neural network FC_2 to generate the output of the graph convolution for the vertices $\{f'''_{v_i}\}$. The features of the arrows $\{f'_{a_i}\}$ are output directly from the first network FC_1.

Every fully-connected layer is followed by an optional dropout layer, and the output is processed by an activation function σ.

Parameters of this layer are the input Dimension D_{in}, the hidden dimension D_{hidden} and the output dimension D_{out}. We will use the shorthand $(D_{\text{in}}, D_{\text{hidden}}, D_{\text{out}})$ to describe the dimensions going forward.

Graph convolution networks as denoted GCN above are then stacked layers of the described GC layer.

4.3.6 Parameters

We use the same parameters as described in the 'Neural Design Network' supplementary material [21] with minor differences. For GC, we present the dimensions of each layer with $(D_{\text{in}}, D_{\text{hidden}}, D_{\text{out}})$, and for fully connected layers FC as $(D_{\text{in}}, D_{\text{out}})$, where $D_{\text{in}}, D_{\text{out}}$ denote the input and output dimensions correspondingly.

Relation module In the relation module, the embedding dimension is $D_{\text{embed}} = 128$ and the dimension of the latent variable is $D_{z_{\text{rel}}} = 32$. GCN_{pred} consists of three graph convolutional layers as detailed in Table 4.1. MLP_{pred} consists of two fully-connected layers as shown in Table 4.2.

The embedding dimension of the ground-truth encoding is $D^{\text{enc}}_{\text{embed}} = 64$, and GCN_{enc} is a three-layered graph convolutional network as described in Table 4.3. MLP_{enc} is a two-headed network shown in Table 4.4.

	GCN_{pred}	σ	d
1	GC(128+32, 512, 128)	ReLU	0.1
2	GC(128, 512, 128)	ReLU	0.1
3	GC(128, 128, 128)	ReLU	0.1

Table 4.1: GCN_{pred} layers.

	MLP_{pred}	σ	d
1	FC(128, 512)	ReLU	0.1
2	FC(512, $\lvert R \rvert$)	softmax	0

Table 4.2: MLP_{pred} layers.

We use the loss weights $\lambda_{\text{rec}_1} = 1$ and $\lambda_{\text{kl}_1} = 0.005$.

	GCN_{enc}	σ	d
1	GC(64, 512, 128)	ReLU	0.1
2	GC(128, 512, 128)	ReLU	0.1
3	GC(128, 128, 128)	ReLU	0.1

Table 4.3: GCN_{enc} layers.

	MLP_{enc}	σ	d
1_μ	FC(128, 32)	-	0
1_σ	FC(128, 32)	-	0

Table 4.4: MLP_{enc} layers.

	GCN_{l1}	σ	d
1	GC(128, 512, 124)	ReLU	0.1
2	GC(124, 512, 124)	ReLU	0.1
3	GC(124, 512, 124)	ReLU	0.1

Table 4.5: GCN_{l1} layers.

	GCN_{l2}	σ	d
1	GC(124+4, 512, 128)	ReLU	0.1
2	GC(128, 512, 128)	ReLU	0.1
3	GC(128, 512, 128)	ReLU	0.1

Table 4.6: GCN_{l2} layers.

	h^{enc}	σ	d
1	FC(4+128, 128)	ReLU	0.1
2	FC(128, 128)	ReLU	0.1
3_μ	FC(128, 32)	-	0
3_σ	FC(128, 32)	-	0

Table 4.7: h^{enc} layers.

	h^{dec}	σ	d
1	FC(32+128, 128)	ReLU	0.1
2	FC(128, 64)	ReLU	0.1
3	FC(64, 4)	-	0

Table 4.8: h^{dec} layers.

Layout module The dimension of the embedding is the same as above with $D_{\text{embed}} = 128$. $\text{GCN}_{\text{l1}}, \text{GCN}_{\text{l2}}$ consist of three layers according to the definitions in Table 4.5 and Table 4.6.

The variational auto-encoder networks are defined as follows: h^{enc} is a network of fully-connected layers, followed by a two-headed output of 32 again as shown in Table 4.7. h^{dec} is a simple fully-connected network with three layers as detailed in Table 4.8. The dimension of the latent code in this module is correspondingly $D_{z_{\text{lay}}} = 32$.

The employed loss weights of this module are $\lambda_{\text{rec}_2} = 1$, $\lambda_{\text{rec}_3} = 10$ (to prioritize size reconstruction), and $\lambda_{\text{kl}_2} = 0.01$.

Refinement module The embedding dimension in the refinement module are given as $D^{\text{refine}}_{\text{embed}_V} = 60$ and $D^{\text{refine}}_{\text{embed}_A} = 64$. $\text{GCN}_{\text{refine}}$ is a graph convolution network with three layers according to Table 4.9. Finally, $\text{MLP}_{\text{refine}}$ is a network of two fully-connected layers as shown in Table 4.10.

	$\text{GCN}_{\text{refine}}$	σ	d
1	GC(64, 512, 128)	ReLU	0.1
2	GC(128, 512, 128)	ReLU	0.1
3	GC(128, 128, 128)	ReLU	0.1

Table 4.9: $\text{GCN}_{\text{refine}}$ layers.

	$\text{MLP}_{\text{refine}}$	σ	d
1	FC(128, 512)	ReLU	0.1
2	FC(512, 4)	-	0

Table 4.10: $\text{MLP}_{\text{refine}}$ layers.

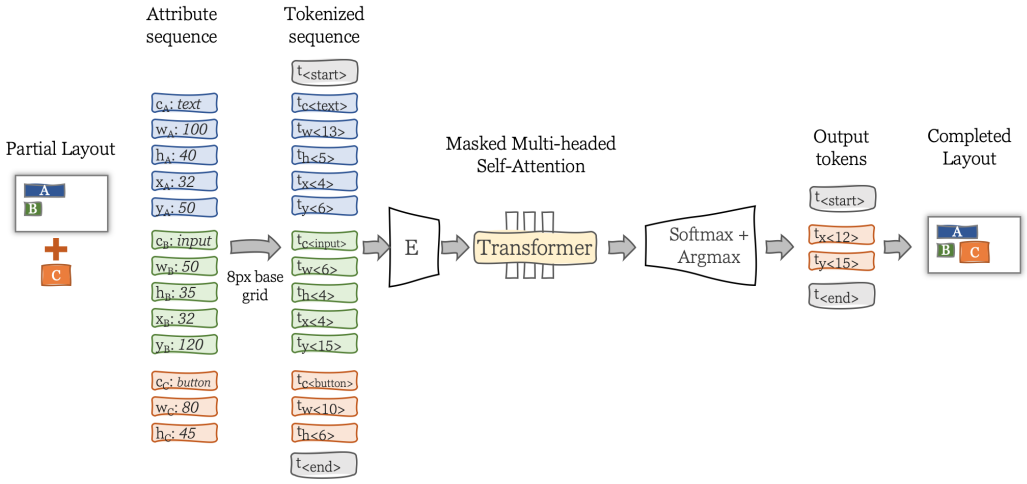

Figure 4.7: The transformer network, modeled after [11].

Training parameters In all fully-connected layers, we add a batch normalization layer, and a dropout layer with a rate of 0.1. We use the optimizer Adam with a learning rate of 10^{-4}, $\beta = (.9, .999)$ and an l2 regularization of 10^{-4}.

We also tried using only the relation module to produce constraints for a combinatorial optimization system. However, the predicted edges were very often not completely consistent such that considering all, no feasible result was possible.

4.4 Transformer-based prediction

Transformers [44] has become a popular neural network architecture choice for many problems, especially in language problems and with other sequential data. Two recent papers also proposed a transformer-based model to address layout completion: Li *et al.* [25] and Gupta *et al.* [11]. Here, we use a similar approach to Gupta *et al.* as it is closer to our stated problem and allows giving constraints on prediction more easily.

4.4.1 Token representation of layouts

In this model, layouts are decomposed into sequences of elements as described in subsection 4.2.1. The input is a partial layout L and a new element e_{new} to be added to it with given element type $c_{e_{\text{new}}}$ and size $w_{e_{\text{new}}}, h_{e_{\text{new}}}$. The

result of the transformer network is a token sequence that can be decoded to decide the positions $\hat{x}^0_{e_{\text{new}}}, \hat{y}^0_{e_{\text{new}}}$.

Instead of combining the properties of an element to form a single embedding for the transformer network, the element properties are interpreted as individual tokens that are embedded separately. This allows to easily condition the network on the new element type and sizes. On the other hand, it produces the challenge of returning valid sequences, as every position in a token sequence then has a particular meaning.

A single element e is represented as $(c_e, w_e, h_e, x^0_e, y^0_e)$. The width and height of an element are put before the coordinates to allow to predefine the desired size of the new element as we defined in our problem statement. A complete layout L is then represented by the concatenation of all element attributes in a flat sequence S_L:

$$S_L = c_{e_0}, w_{e_0}, h_{e_0}, x^0_{e_0}, y^0_{e_0}, ..., c_{e_n}, w_{e_n}, h_{e_n}, x^0_{e_n}, y^0_{e_n}. \tag{4.37}$$

The sequence for an input query with the new element then appends $(c_{e_{\text{new}}}, w_{e_{\text{new}}}, h_{e_{\text{new}}})$ to produce the complete input representation for this model.

Since tokens need to be embedded in the network, we limit the amount of coordinate and size tokens by employing a base grid size g onto which all positions and sizes are "snapped" to. This is actually a common approach in user interface design and automatically prevents misalignments by few pixels. Consequently, all w, h, x^0, y^0 in the layout sequence are divided by g and rounded: $w' = \lfloor \frac{w}{g} \rceil, h' = \lfloor \frac{h}{g} \rceil, x^{0\prime} = \lfloor \frac{x^0}{g} \rceil, y^{0\prime} = \lfloor \frac{y^0}{g} \rceil$, where $\lfloor \cdot \rceil$ denotes a rounding function to the nearest integer.

This updated layout sequence is then converted into a token sequence S^{in}_L for the transformer network. For that, a token dictionary is created with specifically allocated ranges for the different types of element attributes, as well as special transformer tokens for *start*, *end*, and *padding*.

The token vocabulary Vocab is then the following set:

$$\text{Vocab} = (t_{<\text{pad}>}, t_{<\text{start}>}, t_{<\text{end}>}, \{t_{c_i}\}, \{t_{x'_i}\}, \{t_{y'_i}\}, \{t_{w'_i}\}, \{t_{h'_i}\}), \tag{4.38}$$

where $\{t_{c_i}\}$ is the token set of the different component types (i.e., every component type is assigned a token), $\{t_{x'_i}\}$ is the set of tokens for all possible x-coordinates as given by the base grid, and $\{t_{y'_i}\}, \{t_{w'_i}\}, \{t_{h'_i}\}$ are defined correspondingly for the other grid-adjusted element attributes. This requires a maximum canvas size to be defined $W_{\max}, H_{\max}$ that determines the maximum number of tokens for the positions and sizes.

With this token vocabulary, each element attribute is mapped to the corresponding token, i.e., the token representation of an element e is given by

$t_e = t_{c_e}, t_{w'_e}, t_{h'_e}, t_{x^{0'}_e}, t_{y^{0'}_e}$. The input to the transformer model is encapsulated in the special start and stop tokens, and looks then as follows:

$$S_L^{\text{in}} = t_{<\text{start}>}, t_{c_{e_0}}, t_{w'_{e_0}}, t_{h'_{e_0}}, t_{x^{0'}_{e_0}}, t_{y^{0'}_{e_0}}, ..., t_{c_{e_{\text{new}}}}, t_{w'_{e_{\text{new}}}}, t_{h'_{e_{\text{new}}}}, t_{<\text{end}>}. \quad (4.39)$$

4.4.2 Model architecture

The model follows a standard transformer setup [44] and the data flow is depicted in Figure 4.7.

It is composed of stacked encoders and decoders that consist mainly of multi-head attention layers and feed-forward layers. They take in embedded token sequences with explicit positional information so that the order of the input can be understood. In the decoder, the input must be partially masked to prevent reverse information flow of the expected output of the sequences. The final step of the decoder is passing the result through a linear transformation followed by a *softmax* activation function to generate a probability distribution of the next token prediction.

Every batch that is passed through the network must be of the same length, hence, shorter sequences are padded with the special $t_{<\text{pad}>}$ token at the end. Across different batches, different lengths are supported though.

We employ similar parameters in our implementation as proposed by Gupta *et al.* [11]. The embedding dimensions is $D_{\text{embed}} = 512$, the number of layers per encoder/decoder is $n_{\text{layers}} = 6$, with $n_{\text{heads}} = 8$ attention heads, and $D_{\text{ff}} = 1024$ the dimension of units in the feed-forward layers, with the $ReLU$ activation function in both the attention and feed-forward layers.

4.4.3 Training

We specialize our transformer network on the single element prediction task of our problem statement. For that, we generate the training data that only contains the single next new element to be added to the design, as opposed to longer sequences.

We train the network with the cross-entropy loss $CE = -\sum_n^T y_n \log \hat{y}_n$ between the generated token probability and the ground-truth token with Label Smoothing of strength l [39] where $T = |\text{Vocab}|$ is the token vocabulary size, and $y_n, \hat{y}_n$ denote the ground truth probability and predicted probability respectively of token n. Label smoothing puts a high probability on the ground truth category and distributes a small probability uniformly on the other categories. Using such soft targets has been shown to improve the learning and generalization capabilities of neural network models [27].

As in [11], we use the optimizer *AdamW* with parameters lr, β that employs learning rate scheduling such that first a warmup schedule on the learning rate is employed, followed by a decay schedule of the learning rate [26].

In addition, a dropout layer with a rate of d is added after every layer to counter overfitting.

We use a label smoothing strength of $l = 0.1$, a learning rate of $lr = 10^{-4}$ with $\beta = (0.9, 0.999)$, and a dropout rate of $d = 0.1$.

4.4.4 Prediction and decoding

Since a single element is decomposed into 5 distinct attributes in the token sequence that appear at specific positions, we need to control the prediction or decoding of an input such that valid sequences are produced.

Specifically, in our problem statement, we add a partially defined element to the end of an input sequences, i.e., the attributes $(c_{e_{\text{new}}}, w'_{e_{\text{new}}}, h'_{e_{\text{new}}})$, and require a prediction on the missing coordinate attributes $\hat{x}^{0'}_{e_{\text{new}}}, \hat{y}^{0'}_{e_{\text{new}}}$. For that, we can only consider the output probability of the corresponding token ranges $\{t_{x'}\}, \{t_{y'}\}$.

Thus, to ensure valid results, we restrict the token prediction to the expected range for the current position in the sequence and disregard other token probabilities.

Finally, we use a *beam search* to be able to generate diverse variations for the same input, and a temperature parameter *temp* that modifies the output probabilities according to a Boltzmann distribution which allows controlling the 'creativity' of the model (a low temperature amplifies high probabilities, while a high temperature levels all probabilities). In our trials, we used a temperature of $temp = 0.2$ that results in a slightly conservative prediction of the model.

4.5 Sequence alignment with nearest neighbors

Achieving consistency with existing designs naturally requires attending to the patterns in those designs. A straightforward approach to this is a k-nearest neighbor classification or nearest neighbor search. It is a case of instance-based learning in which new problems are compared to the instances in the training data instead of building generalizations from the training data.

Three challenges exist when applying a nearest neighbor search to the stated problem. (1) What representation of layouts to use that allows efficient

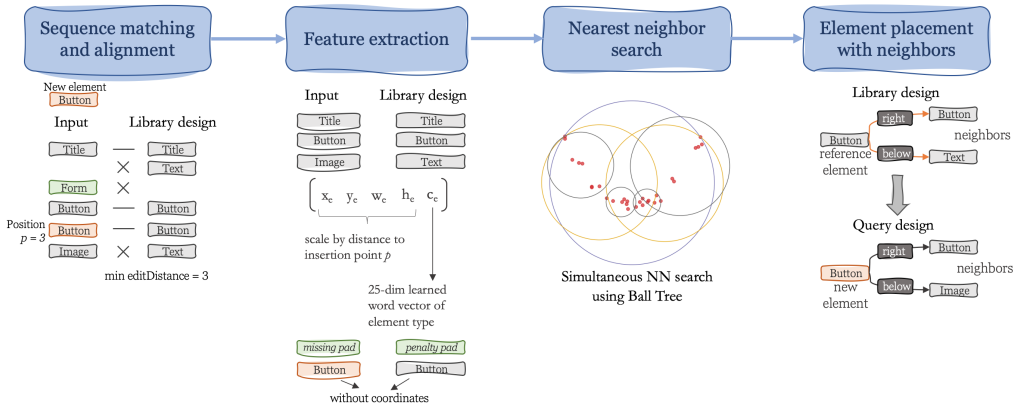

Figure 4.8: Process overview of the sequence-aligned nearest neighbor search with neighborhood placement of the new element.

comparisons and computation of distances in the nearest neighbor algorithm, (2) how to handle the actual placement query for the new element, and (3) how to produce a final layout based on a neighbor from the training data. These will be explored in the next subsections.

4.5.1 Overview

The process is visualized in Figure 4.8. We model the placement problem as finding the best insertion position in the target layout sequence such that the distance to the layout sequences in the design library is minimized.

Given a target layout L^t with n existing elements, there are $n+1$ possible insertion positions in the sequence. For a layout L^{l_i} in the library that also has $n+1$ elements it might suffice to compare the distance between it and the $n+1$ target sequence candidates to determine the best placement with L^{l_i} as a reference. However, as library layouts can be bigger or smaller than the target layout, this does not hold in general. For large differences in layout sizes (e.g., when there are just a few elements already present in the target), there is a combinatorially large number of possible comparison options, even when modeled as a sequence.

Hence, we aim to reduce the number of candidates for comparison by first identifying the best sequence alignment paths between the target layout and a library layout based only on the element types. The overall process is as follows: (1) For every library layout, find the insertion point in the target sequence such that the sequence alignment of the element types is maximized. (2) Create comparison features for all aligned layout sequences and run the nearest neighbor algorithm to find the items with the lowest

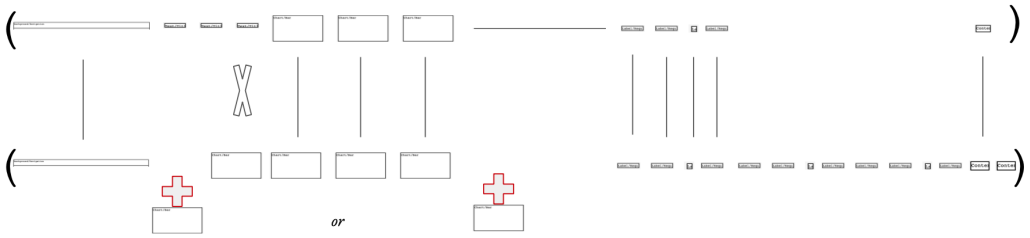

Figure 4.9: Schema of the sequence alignment concept. The bottom element is to be added to the bottom layout sequence. Possible insertion points are determined by aligning the layout sequences with different insertion points and measuring the edit distance.

distance. (3) Place the new element on the canvas in the target design based on the relative position of the neighborhood of the matched element in the reference library layout.

The time and space complexity for the complete method is between $\Theta(\mathbf{N}|L_{\max}|^2)$ and $\Omega(\mathbf{N}|L_{\max}|^3)$ where $\mathbf{N} = |\{L^{l_i}\}|$ is the number of library designs and $|L_{\max}|$ is the maximum number of elements any layout (library or query) contains.

4.5.2 Insertion point and sequence alignment

To determine the closest library designs that guide the placement of a new element, the library layout sequences have to be compared to the target layout sequence with the new element.

One could simplify the problem by allowing only insertions at the end of sequences, similar to text completion problems. Still, given longer library sequences than the target sequence, the challenge remains to identify which elements in the library layout to compare to which elements in the target layout. For this purpose, we use a sequence alignment algorithm that operates on the types of the elements S^{type} (i.e., 'button', 'text', etc.).

With the same approach, the problem can be extended to allow insertion at any arbitrary point in the sequence. Multiple target sequences with different insertion points of the new element can be constructed and the alignment score to a library layout can be calculated. The alignment score is simply the edit distance between the two sequences, so a lower edit distance indicates a better alignment. Figure 4.9 shows an example of how two layout sequences might be aligned with different insertion points that have the same alignment score. This procedure gives a mapping between the library and the target layout that enables to construct the feature representations for the nearest

neighbor search.

Since calculating the alignment paths between sequences is computationally more expensive than only calculating the edit distance, we first find the best insertion positions of the new element by taking those positions that minimize the edit distance between the two sequences.

The candidate positions of the target element p_t^* are thus defined by:

$$p_t^* = \arg \min_{p \in [0, n_t]} \text{editDistance}(S_{t_p}^{\text{type}}, S_{l_i}^{\text{type}}), \tag{4.40}$$

where $p \in [0, n_t]$ are the possible positions in the target type sequence S_t^{type}, $S_{t_p}^{\text{type}}$ represents the target sequence with the new element inserted at position p, $S_{l_i}^{\text{type}}$ is the library type sequence and editDistance is a function that calculates the edit distance between two sequences. For the editDistance algorithm, we utilize an implementation of the Myers's bit vector algorithm [28].

For the remaining target candidates with minimum edit distance, we calculate the sequence alignment path according to Hirschberg's algorithm [14] using [47]. The alignment path then allows to construct a pair of target $\tilde{S}_{t_p}$ and library $\tilde{S}_{l_{i_p}}$ element subsequences for every library item. In these subsequences, the type matches are maximized and features for the nearest neighbor search can be constructed from.

Since the target sequence may be a prefix of a library sequence or it may be a subsequence with "holes" of a library item, different matching strategies might be most appropriate. In the first case, determining the best alignment prefix yields the best results. In such an alignment prefix, removing elements from the library sequence to achieve alignment is not penalized. In the latter case, when sequences are similarly long, it might be more appropriate to test the alignment of the complete sequences. Hence, we calculate both the prefix and full edit distance and alignment paths between the target and library sequences.

4.5.3 Feature representation

Target sequences are reduced to contain no duplicates. Then, every library sequence $\tilde{S}_{l_i}$ and target sequence $\tilde{S}_t$ is then converted to a flat feature representation that can be used as input to a nearest neighbor algorithm.

Every element is represented by its *top-left* coordinate, its *width* and *height*, as well as a 25-dimensional word vector of its *type*. For the new element and the corresponding mapped element in the library sequence, only a representation of *width*, *height*, and *area* is used as the position has not been

calculated yet and the type similarity is ensured via the sequence alignment algorithm.

We use the top-left coordinate of an element and its size instead of a complete bounding box of top-left and bottom-right coordinate to better handle shifted subsequences in which sizes might be very similar but actual positions are shifted by a fixed value. Using sizes ensures that the similarity between such sequences can be found whereas with full bounding boxes there would be a large distance for every point.

More formally, every element e in the layout is represented by its feature f_e:

$$f_e = (\tilde{x}_e^0, \tilde{y}_e^0, \tilde{w}_e, \tilde{h}_e, v_{c_e}), \tag{4.41}$$

where $\tilde{x}_e^0, \tilde{y}_e^0$ are the standardized coordinates of the top-left corner, $\tilde{w}_e, \tilde{h}_e$ the standardized width and height of the element, and $v_{c_e} \in \mathbb{R}^{25}$ the word vector of the component name.

The word vector encodes the component type and allows calculating a type distance between elements. For similar concepts, such as 'label' and 'text', the vector representations have a smaller distance, while unrelated concepts such as 'button' and 'table' are separated by a longer distance. We use *fastText*'s pretrained word representations that are trained on Common Crawl[1] and Wikipedia text [10].

The basic form of the layout sequence $\tilde{S}$ is then a concatenation of the individual element features:

$$\tilde{S} = f_{e_{p_0}}, f_{e_{p_1}}, ..., f_{e_{p_n}}, \tag{4.42}$$

where $f_{e_{p_i}}$ is the feature of the element in the ith position in the aligned sequence. All features are "unwrapped" to prevent a 2-dimensional vector and ensure that a 1-dimensional feature representation is created.

Not all elements of a layout might be directly matched between the target and library layout. For example, if the library layout contains more elements than the target layout with the new element, some elements in the library layout will not be encoded in the feature representation.

Similarly, the new element of the target layout does not have coordinates yet. Hence, only its dimensions and area are used to encode it: $f_{e_t} = (\tilde{w}_{e_t}, \tilde{h}_{e_t}, \alpha\sqrt{\tilde{w}_{e_t}\tilde{h}_{e_t}}, v_{c_{e_t}})$. We want to place a high emphasis on finding a placement for the new element where the reference element in the library layout has a similar size. Thus, in addition to the width and height of the new element, also a scaled value of the area is added to its feature representation. To keep it in a similar dimension as the individual coordinates, the

[1] https://commoncrawl.org/

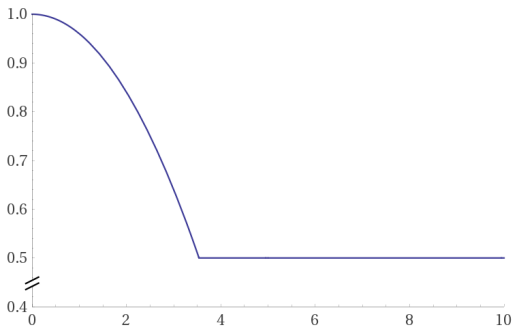

Figure 4.10: The scaling function $t(p_e, p_t)$ to emphasize local similarity, with $\lambda_{\min} = 0.5$ and $\gamma = 5$. The x-axis shows the absolute distance from the target position $|p_e - p_t|$. The scaling factor for close elements is high that gradually decreases with increasing speed until it reaches the minimum scaling factor.

square root of the area is calculated. This applies both to the new element in the target layout e_t as well as the matched element e_{p^*} in the library layout according to the sequence alignment result. With the scaling factor α, one can increase or decrease the size matching requirements for the new element.

Further, element coordinates and sizes are standardized to the canvas size, so that it ranges from $[0, 1]$. This is needed in case the library layouts do not have the exact same canvas size and to keep it in the same order as the word vectors that are in the range $(-1, 1)$.

In addition, we argue that elements that are closer to the insertion point in the sequence are more relevant than elements that are farther. To accommodate this, we scale each feature vector by its distance to the insertion point of the new element $t(p_e, p_t)f_e$ where $t(.,.)$ is a function returning the scaling factor between the position of the current element p_e and the position of the target element p_t. For $t(.,.)$ we use the following formula:

$$t(p_e, p_t) = \max(\lambda_{\min}, 1 - \min(1, \frac{|p_t - p_e|^2}{\gamma^2})), \tag{4.43}$$

where $\lambda_{\min}$ is the minimum scaling factor for far away elements, and γ is the distance at which point the minimum scaling takes effect. This function is rendered in Figure 4.10.

The last property of our similarity search follows the idea that the more elements that can be matched, the better. Since the sequence alignment algorithm finds the best matching component type subsequence, it could suggest a very small overlap, that could lead to a small distance if considered on its own. To counter this effect, every element from the target layout that is not matched is added as a penalty feature f_e^p to the target layout. The

penalty feature is composed of the distance of the element to an approximate location of the new element and its size:

$$f_e^p = (\tilde{\delta}\tilde{x}_e, \tilde{\delta}\tilde{y}_e, \tilde{w}_e, \tilde{h}_e). \tag{4.44}$$

The distance to the new element is approximated since the final location is not known yet. For this, the center position of the elements surrounding the insertion point is taken as an approximation for the new element, and the distance to the center point of the missing element is calculated:

$$\tilde{\delta}\tilde{x}_e = |\frac{(\tilde{x}^0_{e_{p^*-1}} + \frac{\tilde{w}_{e_{p^*-1}}}{2}) + (\tilde{x}^0_{e_{p^*+1}} + \frac{\tilde{w}_{e_{p^*+1}}}{2})}{2} - (\tilde{x}^0_e + \frac{\tilde{w}_e}{2})|, \tag{4.45}$$

where e_{p^*-1}, e_{p^*+1} are the elements before and after the insertion point in the sequence. The same formula can be applied to the y-coordinate by replacing all $\tilde{x}_e$ with $\tilde{y}_e$.

Listing 4.1: Example feature representation of a 6+1 element layout search.

```
# box features of matched elements in the layout
(0.02,  0.03,  0.21,  0.03,
 0.03,  0.10,  0.79,  0.18,
 0.03,  0.35,  0.11,  0.02,
 0.03,  0.40,  0.45,  0.08,
 0.03,  0.51,  0.37,  0.22,
# box padding to match longest feature sequence
100.0, 100.0, 100.0, 100.0,
# box feature of target element
 0.14,  0.04,  3.59,

# word vector of the component names
 0.02, -0.01,  0.00, ...,  0.09, -0.02,  0.05, -0.04, -0.03,
 0.03, -0.11, -0.02, ...,  0.08,  0.01,  0.05,  0.03, -0.00,
-0.01, -0.13,  0.04, ...,  0.13, -0.01,  0.16,  0.02,  0.04,
 0.17, -0.19,  0.00, ..., -0.02,  0.07,  0.04,  0.01,  0.03,
 0.07, -0.11,  0.03, ...,  0.02,  0.02,  0.17, -0.10, -0.01,
# word vector padding
10.00, 10.00, 10.00, ..., 10.00, 10.00, 10.00, 10.00, 10.00,
# word vector of target element
-0.03, -0.17,  0.15, ...,  0.13,  0.00,  0.05, -0.03,  0.02)
```

Finally, the approximate distance is inverted so that close elements that are not mapped produce a higher penalty than those that are far away. The following formula is applied:

$$\tilde{\tilde{\delta}}\tilde{x}_e = \max(0, 0.5 - \tilde{\delta}\tilde{x}_e), \tag{4.46}$$

$$\tilde{\delta}\tilde{y}_e = 0.5 \max(0, 0.5 - \tilde{\delta}\tilde{y}_e). \tag{4.47}$$

The vertical distance is scaled down because we argue that missing elements in the same row are more relevant than those in other rows even if the horizontal distance is less in the second case.

Lastly, the penalty feature is scaled with the same function $t(p_e, p_t)$ but with different parameters λ_{min} and γ.

The dimension of the feature vector is thus a function of the number of maximally matched elements n^* (incl. the new element) in the sequence: $|\tilde{S}| = 4(n^* - 1) + 3 + 25n^*$. Listing 4.1 shows an example of a representation of a layout sequence where 6 elements were matched in the same sequence while the longest sequence match contained 7 elements. As a result, the total dimension of the feature vector is 202 (word vectors are cropped to fit on the page). The padding is explained in the next subsection.

4.5.4 Nearest neighbor search

As the number of elements in the layouts may vary, it could happen that when compared to the target layout, the length of the subsequence match differs between layouts. This would result in a differently sized feature vector. To optimize the search algorithm so that only a single call to the nearest neighbor algorithm is needed, all feature vectors need to be of the same dimension. Hence, smaller sequence representations are padded with dummy values for the missing elements so that all sequences have the same feature dimension (x^{dummy} for box features and v^{dummy} for word vectors).

Since all of the target sequences and library sequences are used in the same nearest neighbor search for performance reasons, the different parts of the feature vector are assigned different numerical regions so that comparisons between differently sized sequences incur a high distance. The regular features of the elements that are mapped have dimensions around $(0, 1)$. Penalty features are offset by $o^{\text{penalty}} = 100$. Similarly, the padding offset is $o^{\text{pad}} = -100$, and $v^{\text{pad}} = -10$.

To be able to use efficient data structures for a fast nearest neighbor search, such as Ball Tree or KDTree, the features must be comparable by a single true metric that exhibits the properties of identity, symmetry, and triangle inequality. While it is more common to use cosine similarity for word similarity, it has the disadvantage of not exhibiting the triangle inequality, and using different distance functions for different parts of the feature vectors would also prevent using the efficient nearest neighbor algorithm structures as implemented in Scikit-Learn.

Finally, the desired number of neighbors is increased in the neighbor

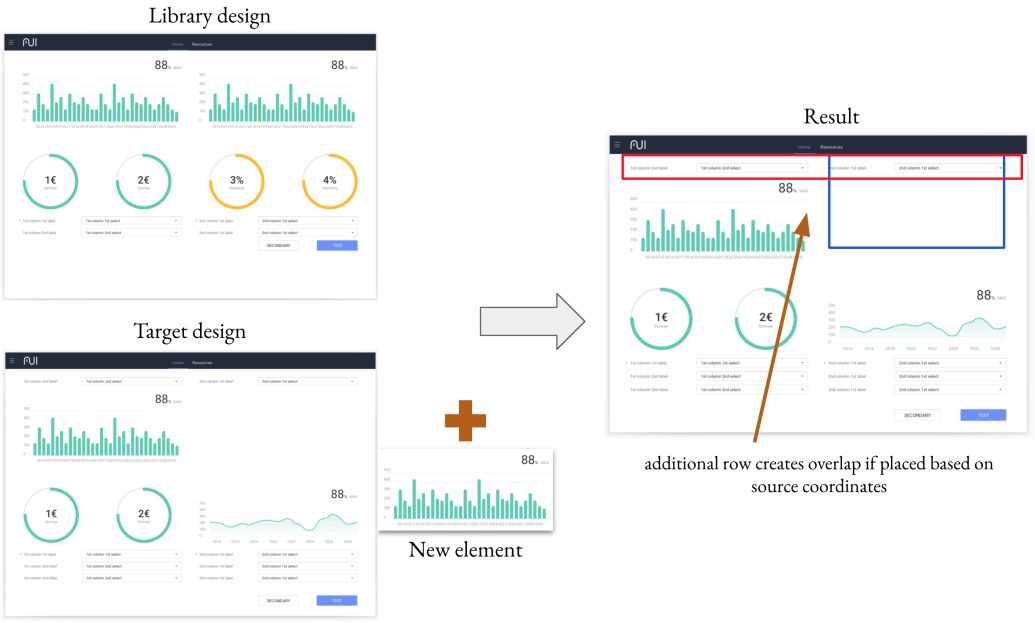

Figure 4.11: Naive placement of an element according to a reference design without taking differences and shifted patterns into account.

search by a factor $\eta = 3$ to account for the effect that a resulting match might not produce a feasible layout. The procedure of constructing a placement for the new element from the sequence neighbor is explained in the next section.

4.5.5 Producing final layouts

Once a neighbor for the query layout is identified, the new element can be placed on the query layout. One could try to place the element in the same position as the mapped element in the neighbor layout. However, this is likely to fail if the overall structure between the layouts has larger differences. As an example, if the query layout has an additional row of elements so that all elements below are shifted to the bottom, copying the coordinates will produce an overlap with existing elements. This is visualized in Figure 4.11.

Instead, we consider the *neighborhood* of the mapped element in the neighbor layout and try to apply the relative positioning rules to the query layout. That way, shifted elements are not posing a direct issue.

Specifically, we consider the neighborhood of the mapped target element in the neighbor layout as the set of elements that are direct neighbors in any of the four directions (above, below, left, right) of the element. Direct neighbor means that no other element is between the connecting line between

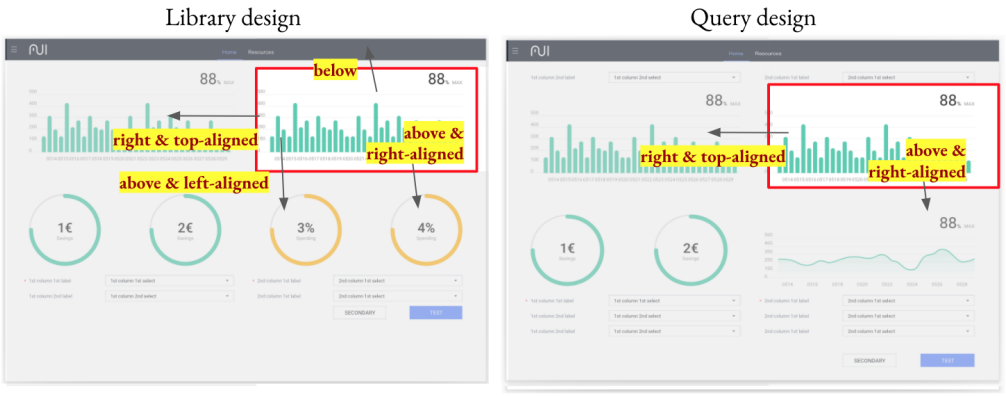

Figure 4.12: Placement approach using relations to neighbor elements.

those elements.

Of this neighborhood, we consider the graph representation as detailed in subsection 4.2.2, i.e., the relations in regards to position and alignment, to the mapped target element. These relations are then applied to the query layout, using the mapping between the elements produced by the sequence alignment algorithm. Neighbor elements that are not mapped are ignored. This is shown in Figure 4.12.

If the resulting placement creates an invalid layout because the element is overlapping with another element or is placed outside the canvas, the relations are successively reduced according to the distance to the target element such that farther neighbors are ignored for the subsequent layout trial. If no relaxation creates a good layout, the candidate is discarded. The resulting layouts are finally presented to the user as placement suggestions, sorted by the distance computed in the nearest neighbor search.

Producing the layout is not trivial, as it requires identifying a good placement according to possibly competing relations. While it might be desirable to create final layouts for all neighbor sequences as identified in the sequence alignment step, it would incur a high runtime cost, and thus, it is performed as the last step only for a small number of good candidates.

To enable interaction with the described methods, we developed a plugin for a UI design software. This is briefly described in the section chapter, along with details on the machine learning implementations.

Chapter 5

Implementation

In this chapter, we describe how the methods were integrated into a user interface design program and give a few details on the technical implementations of the machine learning models.

5.1 Design tool integration

For these methods to be used properly in a design process it must be integrated into a complete user interface design program. For this purpose, we created a plugin for the Sketch software, a popular user interface design tool. Figure 5.1 shows the plugin overlaid over the regular Sketch window.

The plugin is built with web technologies such as the Javascript framework React that connects to a Python backend that executes the requested commands and returns the results. As such it can be deployed more widely in an organization with a central computing infrastructure as all the heavy lifting is done outside of the client-side plugin.

Sketch provides plugins the designs of the current document via an application programming interface, representing the designs in a nested structure as it is shown in the layers area. While layers can be a multitude of different types we require that the layout is primarily composed of well-defined components that are called symbols in Sketch. These symbols represent the components that are available via the design system to the UI designer. Without these symbols, the individual layers and shapes could not easily be identified as to what they represent as at the lowest level they are just composed of geometric shapes.

Layouts are thus converted to a flat list of components with respective properties, such as their type and their coordinates and dimensions. To identify components, we assume a naming convention that allows mapping

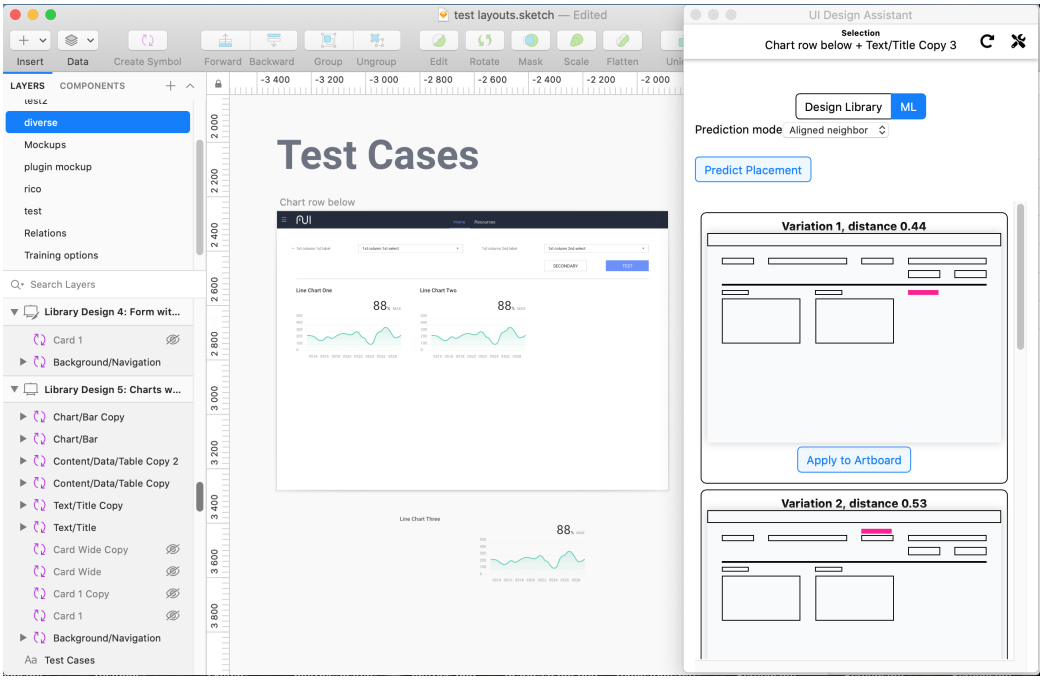

Figure 5.1: The user interface of *Sketch* with our plugin loaded on the right side. It shows the placement suggestion for a new text element onto the artboard in the middle of the screen.

components and their variations to a set of known components. For example, a 'button' component might have a variation with different styles with names such as 'button/primary' and 'button/secondary' but we consider them both to be the same component type 'button'.

As an external plugin to Sketch, the interaction options are slightly limited with respect to creating smooth usability. Hence, the user has to first click on the artboard that he wants to generate a prediction for which is registered in the plugin, and the current component view of the layout is shown as a preview. Next, the new element must be added to an area outside of the canvas and selected which is also recognized by the plugin and displayed. Then, the user can choose the prediction method and request candidates.

A list of candidates is computed in the backend and their JSON representation is returned to the plugin. It generates previews of the suggestions with the new element highlighted in the layout which can be applied to the actual design via a button.

5.2 Machine learning implementations

All methods were implemented with Python 3.8 and were run with a fixed seed to have reproducible results.

The nearest neighbor search uses the library Scikit-learn 0.22.1 that provides efficient implementations of the KDTree and BallTree structure. In addition, it uses the library edlib for the sequence alignment algorithm [47].

For the Transformer model, we used Tensorflow 2.3 with the Keras backend. We take advantage of the predefined Transformer network library "Keras-transformer"[1] for the network definitions. We added a proper beam search and custom decoding to support restricting the token predictions at specific positions.

The graph neural network was programmed with the framework PyTorch 1.4. We use the code of the sg2im project[2] as a foundation for the layers in the graph neural network which was heavily modified to follow the descriptions from "Neural Design Network" [21].

[1] https://github.com/CyberZHG/keras-transformer
[2] https://github.com/google/sg2im

Chapter 6

Evaluation

In this chapter, we discuss the evaluation of the presented methods. We first present the data sets we used for the evaluation. Then, we describe our results on these data sets, first quantitatively, followed by a qualitative inspection of examples.

6.1 Data sets

Our problem statement is aimed towards consistency in layouts and design patterns, however, we did not find any prior data set of layouts that follows a single design system or similar, and thus would exhibit clear commonalities and design patterns. As a result, we created two data sets of our own that have common properties from a single design system. In addition, we also used a variation of the Enrico data set [23], as well as a larger set of Rico layouts [5] in the same fashion as it was done in "Neural Design Network" [21] to test the methods on large in-the-wild designs although there is no common theme or design patterns encoded in that data. We first describe the custom data sets and then detail our usage of Enrico and Rico.

Varying buttons This data set consists of small web-like layouts with 3-7 high-level components (*headline, form, text, button, image, table*). It follows a simple 2 column grid layout with the majority of the elements being positioned in the first column and left-aligned.

Most components are placed in very similar positions if present. The *headline* component is always present and in the same position, while the *form* component is always present with the same x-coordinate always but a different vertical position. The *button* component is always present but is placed at 5 different positions in relation to the other components: (1) below

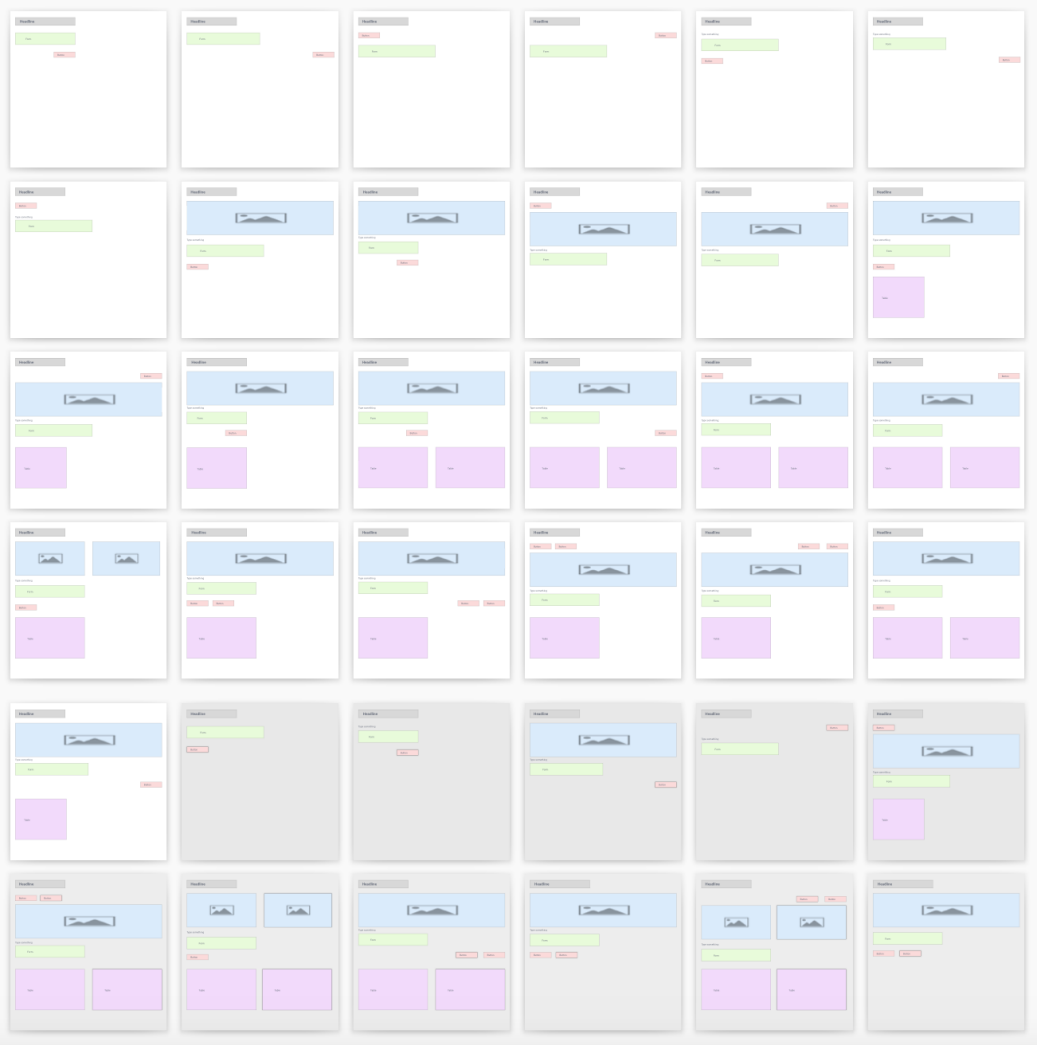

Figure 6.1: The set of layouts in the varying buttons data set. Grey boxes correspond to headlines, green boxes to forms, red boxes to buttons, blue boxes to images, pink boxes to tables and black text represents a text paragraph. Grey layouts indicate test items that are not used during training.

the headline and left-aligned, (2) below the headline and at the right edge of the canvas, (3) below the form and left-aligned, (4) below the form and right-aligned, and (5) below the form and at the right edge of the canvas. This is the main pattern that is encoded in all of the layouts.

Variations of this pattern are created with additional elements added at fixed rows. A *text* or *image* element is added above the form, while a *table* element might be added below the form and the button.

Additionally, a few further variations exist where one component exists twice in the layout. This can be either the *button, image* or *table*. This secondary pattern is such that the row position of the additional element of the same component type does not change, and is always placed next to the other element of the same type.

In total, we created 36 layouts with these properties of which 25 are used as the design library, and 11 layouts are test cases. This set is shown in Figure 6.1 with the design library layouts having a white background and the test cases having a grey background.

When building layouts according to this data set in a sequential matter, the placement of a next element is quite clear as only the button has some valid variations. This is also true when adding elements to an existing but packed layout as for every new type of element there exist clear rules.

On the other hand, this set does not allow generalizing beyond the patterns mentioned above, so it is not possible to derive many further test cases.

Regarding the element relations, it exhibits predominantly vertical positions (*above/below*) with only a few *left/right* cases, and mainly *left-aligned* relations with few *top-aligned* and *at right of canvas* edges.

The main purpose of this data set is to test if different valid variations can be retrieved with a method and to test the general capabilities given relatively fixed positions of the elements in small layouts.

Artificial web layouts While the previous layout set was simple with few variations that make it easy to evaluate predictions for their consistency, it contains too few data points that it can be effectively used for most neural network architectures.

Hence, we created a second, larger set of layouts. It consists of 359 layouts of which 52 belong to the test set. The number of elements ranges from 4 to 38 per layout belonging to one of 18 different component types, so there are often many elements of the same type present. The grid system is typical for traditional, non-application, web-based pages, following a horizontally centered, top-to-bottom column layout. Every design consists of 2-3 sections in addition to a static *navigation bar* and *footer*. Every section is roughly centered horizontally on the page and might consist of multiple columns in itself. The first and optional third sections are a random "filler" section that embeds the specific pattern in a variety of different contexts. These filler sections are, e.g., hero sections typically found in landing pages, call-to-action sections (e.g., for subscribing to a newsletter), or teaser sections with images, teaser texts and, a link with further information (e.g., for news items or feature descriptions). In between, there are three design patterns encoded

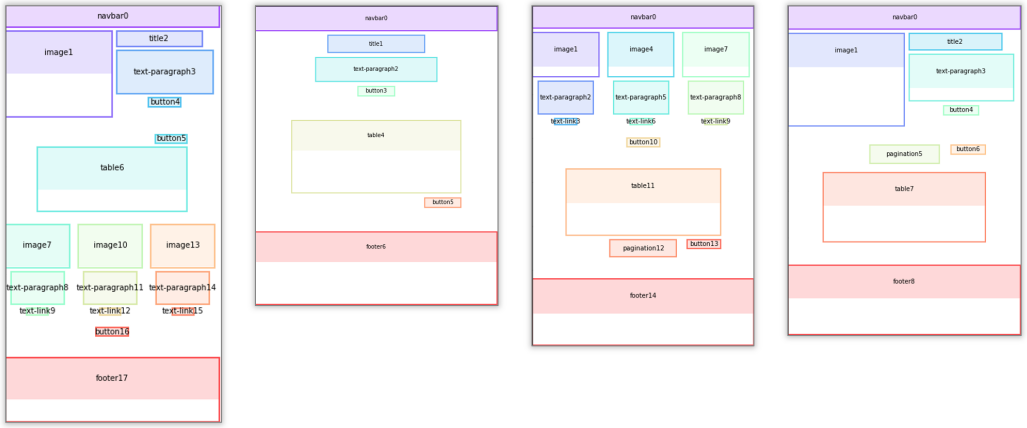

Figure 6.2: The table pattern. The button accompanying the table is always right-aligned with the table and can be either above or below it. If the pagination is present, it is next to it.

that can be typically found in web-based layouts.

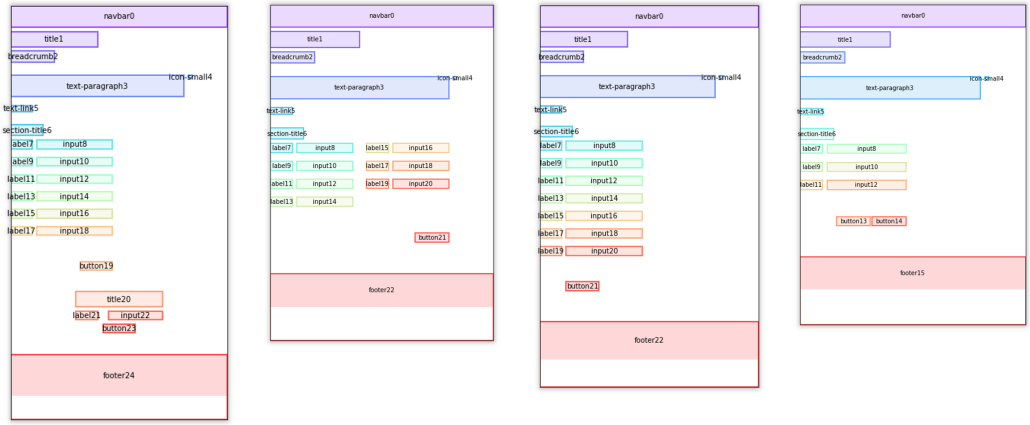

Figure 6.3: The form pattern. There can be forms with one or two columns with the label to the left of the input. A single button can be either placed left or right-aligned with the right-most input field. With two buttons, they are placed next to each other and are right-aligned with the last input element.

(1) Data tables. A data table pattern that combines a *table* component with a *button* component (e.g., to execute an action with a selection of items from the data table). Therein, the button is always aligned to the right of the table but can be either above or below the table. Additionally, there might be

a *pagination* component that is horizontally centered with the table element, and it can be above or below it. If such a pagination element is present, the button is always next to it, i.e., the relative spatial position to the table must be the same for the pagination and button elements. Examples of this pattern are shown in Figure 6.2.

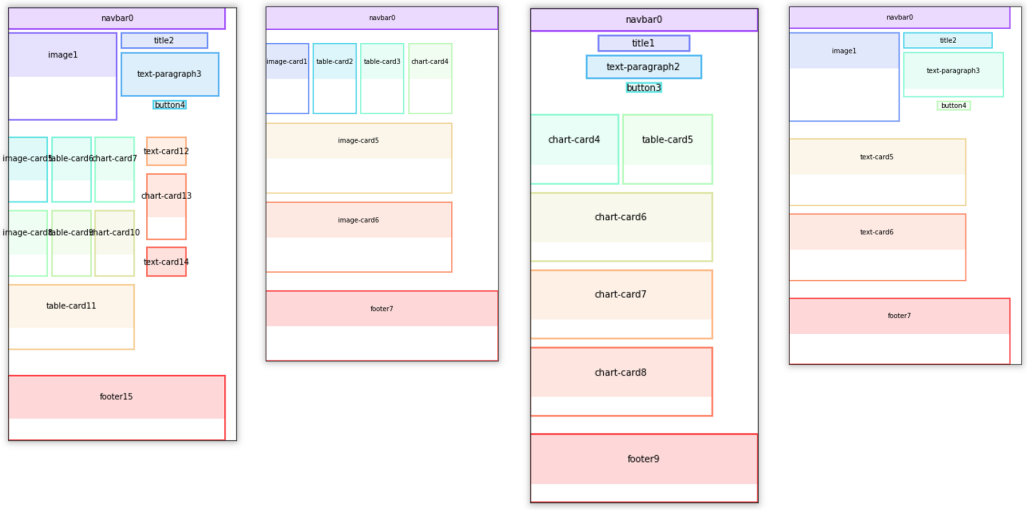

Figure 6.4: The dashboard pattern arranges different type of cards in either a 2-column layout, 4-column layout, or a 3-column layout with an additional sidebar. Cards can either span a single column or all columns.

(2) Forms. In this data set, we follow a common web form pattern where the label is placed to the left of an input element. The complete form is either arranged in a single column (70% of times) or in two columns (30%). The preceding elements are static and always consist of a title, breadcrumb, a text paragraph, an icon, a link, and a section title. After the form, an optional filler section might be present.

As before, the button following the form is forming the main pattern in this context. When there is a single button after the form, it can be aligned to the left of the last input field (20%) or right-aligned (80%). This applies to both column variations. When two buttons are needed after a form, they are placed next to each other in the same row, and the right button is right-aligned with the last input field. This is shown in Figure 6.3.

(3) Dashboards. Lastly, different dashboard layouts are present in the data set. It is composed of at least two rows of different widgets. There

are three different layout arrangements: a two-column layout, a four-column layout, and a three-column layout with a separate sidebar column. Each widget can take the full width of the columns or a single column. There are four different widget types that are used randomly in every layout (text, chart, table, image). The different layouts are present in equal share in the data set. Examples of these are shown in Figure 6.4. For these dashboards, there is an optional filler section above present.

Since there is no data set available with specific patterns encoded in them our goal was to create a controlled set of layouts with known patterns with an extensive number of examples in order to evaluate whether neural networks models are able to pick up these patterns when a larger number of examples is given.

Apart from the described patterns, it also allows testing placement suggestions for the static elements such as the filler sections, navigation bar, and footer. With the different types of widgets in the dashboards, it can further be seen if similarities of these types are identified.

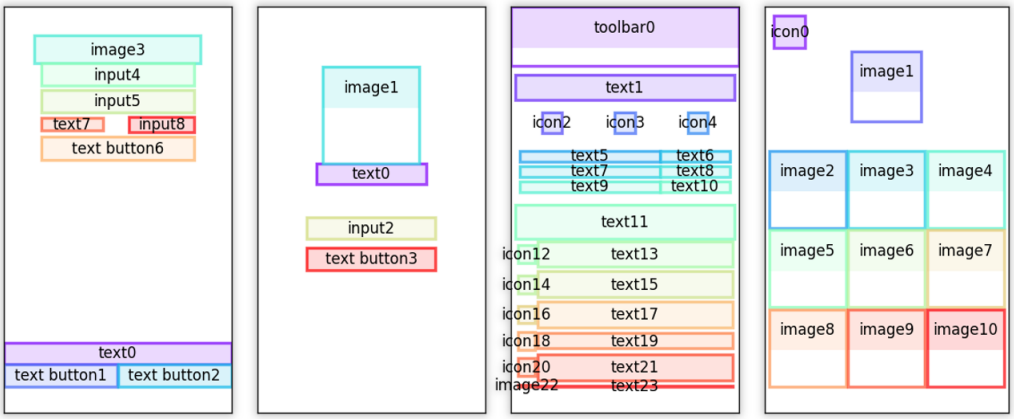

Figure 6.5: Example layouts that are considered high-quality based on Enrico and are compatible with our assumptions (no overlap, valid representations as graphs and sequences).

Enrico Since our premise is to predict element placements based on patterns in the data, the in-the-wild layouts from Rico [5] do not allow to evaluate this directly as there are no clear patterns or commonalities between all of the applications that were used to collect the data set.

Further, recent studies have shown that the quality of the layouts contained in the full set of Rico is of mixed quality with only around 10% of

contained layouts conforming to good layout guidelines [20, 23]. Hence, we evaluate the results on such a subset of "good" layouts as provided by Enrico [23]. Taking a set of layouts of better quality might increase the likelihood of at least some commonalities between the layouts.

We further filter the set of good layouts from Enrico to remove layouts with overlap between elements so that it fits directly our initially declared scope and to keep it in line with the handcrafted data sets. In addition, we ignore layouts that are not compatible with the conversion algorithm that converts layouts into graphs and sequences, as they might encode edge cases that might not be supported by other algorithms down the line either.

In the final set of layouts, there are 766 items with element counts ranging from 3 to 38. As there are no clear patterns to balance training and test data with, we simply take 10% of these layouts as test layouts. Examples of these are shown in Figure 6.5.

Predicting placements according to this data set can then be only evaluated according to general layout principles and similarities to the data set. As we excluded overlap in the layout, one important evaluation criterion is that the placement does not overlap with anything else in the layout.

Rico (NDN)

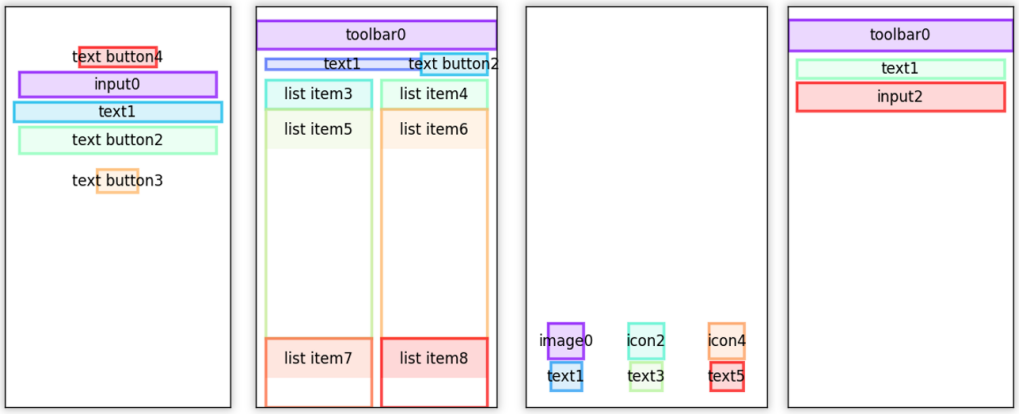

Figure 6.6: Example layouts from the *Rico (NDN)* data set. All layouts have less than 10 elements, hence, the complexity is lower than the *Enrico* set.

Lastly, we test results on an even larger subset of Rico [5] where graph neural networks have been shown to work properly before. For this, we follow the same selection method as in "Neural Design Network" (NDN) [21] and only take layouts with less than 10 elements and only consider layouts that are composed of the most used component types. As such, we disregard

	Layouts			*Queries*	
Data set	**Train**	**Test**	**Elements**	**Train**	**Test**
Varying buttons	25	11	3–9	355	169
Artificial web	307	52	4–38	41,284	4,998
Enrico	689	77	2–38	34,943	2,652
Rico (NDN)	19,401	2,156	2–9	260,046	25,317

Table 6.1: Data set statistics. Queries are constructed by decomposing layouts and placing each element in sequentially growing compositions.

the 6 least used component types (*Checkbox*, *Date Picker*, *Number Stepper*, *Button Bar*, *Map View*, *Video*) that are present less than 100 times and remove all layouts that contain one of them. In addition, we ignore layouts that have only 1 element and those with overlap according to our problem restrictions. This results in 21,557 layouts for the final data set (out of 66,261, 24,262 contain more than 9 elements, 8,084 contain only 1 element, 12,121 of the remaining are with overlap, and 237 contain one of the least used components). As before, we take a random sample of 10% as test data. Figure 6.6 shows example layouts of this data set.

6.2 Training and test sets

To generate the training data, the set of reference designs are taken and decomposed sequentially according to the sequence representation (i.e., scanned in rows from top-left to bottom-right). For example, a vertical layout with three elements ‘headline’, ‘form’, and ‘button’ below each other would generate the sublayouts (‘headline’), (‘headline’, ‘form’) and (‘headline’, ‘form’, ‘button’).

In every sublayout created as such, we generate all possible inputs for our methods by removing every element once and marking it as the new element to be added to the layout.

This produces an exhaustive number of actual inputs for the methods to be used during training and for testing generalization capabilities. These numbers are shown in Table 6.1. For the handcrafted data sets ‘Varying buttons’ and ‘Artificial web’, the test set is carefully created to contain the encoded patterns without overlap in the training data. For ‘Enrico’ and ‘Rico (NDN)’, we randomly sample with a fixed seed 10% of the data set as the test set.

Figure 6.7: Examples of exact and close matches. In the exact match, the alignment to the neighboring element is according to the expectation in green, even if it has a minor offset vertically. The close match has a major overlap but misses the alignment.

6.3 Metrics

Based on our problem statement, we want the element placements to follow the patterns from the design library / training data. As such, predictions should not only follow general layout principles but also be consistent with similar layouts in the data set.

Exact and close match. We compare suggestions to the expectation and measure two levels of matches: exact match and close match as depicted in Figure 6.7.

An *exact match* is achieved if the predicted placement of the element matches the expectation such that the relative positions and alignments to neighboring elements are the same, and the Intersection of Union (IoU) is above a threshold η_{exact}.

We define the set of elements that are neighboring elements of e as K_e. An element e_j is considered a neighbor of e_i if you can draw a straight line from any edge of e_i to e_j without crossing any other element e_k. Additionally, the special *canvas* element is always contained in the neighbor set. Then, we compare the arrows from the element to the neighbors $e \to K_e$ of the graph representation of the predicted layout $\hat{A}_{e \to K_e}$ with the set of arrows in the ground truth graph $A^*_{e \to K_e}$. If all relations $r_i \in \hat{A}_{e \to K_e}$ of the prediction match the relations in the expectation $r_j \in A^*_{e \to K_e}$, the neighborhood is considered equal.

Since this does not take into account the exact position, we check the IoU between the prediction and the expectation as well. We calculate the

IoU between the predicted bounding box $\hat{b}_e$ of an element and the expected bounding box b_e^* according to:

$$\text{IoU}(\hat{b}_e, b_e^*) = \frac{|\hat{b}_e \cap b_e^*|}{|\hat{b}_e \cup b_e^*|}. \tag{6.1}$$

We allow a few pixels difference if the alignment is matched, and require an IoU of $\eta_{\text{exact}} > 0.7$ to be considered an exact match. In practice, this allows, e.g., a small offset of 10 pixels for small elements like buttons.

If that is not achieved, we define the *close match* if the placement has a non-insignificant overlap with the expected placement. We consider this if the IoU is above the threshold of $\eta_{\text{close}} > 0.15$.

Valid results, overlap and outside of canvas. Following our premise of the problem and data, we do not expect any overlap (as there is no overlap in any layout). We use the following formula to determine if there is overlap between two elements e_1, e_2:

$$\min(x_{e_1}^1, x_{e_2}^1) > \max(x_{e_1}^0, x_{e_2}^0) \wedge \min(y_{e_1}^1, y_{e_2}^1) > \max(y_{e_1}^0, y_{e_2}^0). \tag{6.2}$$

Note that we do not consider it overlap if two elements' borders touch each other, as this can be a valid arrangement in graphical layouts. Predictions that are overlapping are considered invalid and are counted as N_{overlap}.

Further, if an element's bounding box extends beyond the canvas size, it is considered invalid as well, as counted in N_{outside}. The following test determines if an element e is outside the canvas of the layout L:

$$\min(x_e^0, y_e^0) < 0 \vee x_e^1 > w_L \vee y_e^1 > h_L. \tag{6.3}$$

All other cases are considered valid results. We calculate the rate of valid results according to:

$$\frac{N_{\text{total}} - N_{\text{overlap}} - N_{\text{outside}}}{N_{\text{total}}}, \tag{6.4}$$

where N_{total} is the number of total results generated.

Alignment. We measure the alignment of the generated element placement to the rest of the layout. For this, we follow a similar approach as described in [21], and calculate the alignment score α as the mean of the horizontal and vertical alignment as the minimum of the distance of any alignment line of the new element e_{new} to any other element e_i in standardized form:

$$\alpha_{e_{new}} = \frac{1}{2} \min_{e_i} \{||\tilde{x^0}_{e_{new}} - \tilde{x^0}_{e_i}||_1, ||\tilde{x^1}_{e_{new}} - \tilde{x^1}_{e_i}||_1, ||\tilde{x^c}_{e_{new}} - \tilde{x^c}_{e_i}||_1\}$$
$$+ \min_{e_i} \{||\tilde{y^0}_{e_{new}} - \tilde{y^0}_{e_i}||_1, ||\tilde{y^1}_{e_{new}} - \tilde{y^1}_{e_i}||_1, ||\tilde{y^c}_{e_{new}} - \tilde{y^c}_{e_i}||_1\}. \quad (6.5)$$

Figure 6.8 shows an example that depicts the minimum differences of both the horizontal and vertical alignment lines to the other elements.

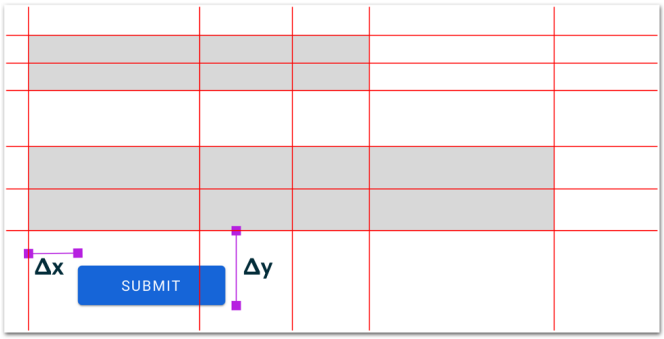

Figure 6.8: Alignment takes the average of the minimum differences between any x^0, x^c, x^1 of the new element and the existing elements (shown with Δx) and between any y^0, y^c, y^1 of the new element and the existing elements (Δy).

Retrieval. Queries that are part of the training data are considered *retrieval queries*. For all data sets, retrieval queries are measured with the *exact* and *close match*. Placing an element in the same way the same composition was done before achieves the highest form of consistency. Additionally, we also measure general layout qualities with the *valid rate, overlap* and *outside counts*.

Generalization. In the handcrafted data sets, we have encoded specific layout patterns. For these, we test *generalization queries* and measure the *exact* and *close match* scores as before, along with the general layout measures *valid rate, overlap* and *outside counts*. For the other data sets with in-the-wild layouts, we do not calculate these match scores because we cannot expect the placements to be learned for unseen queries that are possibly not similar to the training data. Instead, for these, we only measure the general layout qualities with the *valid rate, overlap* and *outside counts*.

6.4 Quantitative results

This section presents the quantitative results for all methods. We evaluate the methods based on the metrics listed above and present the results in separate tables for each data set.

Varying buttons	kNN	Transformer	GNN
Retrieval			
Valid	70.1 %	46.0 %	50.1 %
Overlap	28.4 %	31.5 %	44.2 %
Outside	1.5 %	22.5 %	5.7 %
Alignment	.052 [.028, .085]	.086 [.039, .156]	.065 [.040, .100]
Exact match	99.7 %	61.2 %	0.3 %
Close match	0.0 %	21.6 %	62.9 %
Failure	0.3 %	17.2 %	36.8 %
Generalization			
Valid	68.5 %	46.2 %	46.2 %
Overlap	29.2 %	35.1 %	51.1 %
Outside	2.3 %	18.7 %	2.7 %
Alignment	.046 [.021, .086]	.086 [.039, .164]	.055 [.036, .103]
Exact match	72.4 %	36.5 %	1.1 %
Close match	14.4 %	43.1 %	48.6 %
Failure	13.2 %	20.4 %	50.3 %

Table 6.2: Evaluation statistics for the 'varying buttons' data set. Values in brackets correspond to the 0.25 and 0.75 quantile.

Since a single input can resolve to multiple valid results (e.g., there are multiple valid positions for placing a new *button*), we query 3 suggestions for placement for each prediction task. Increasing the number of suggestions could increase the matching scores, but might at the same time affect negatively the alignment scores if more diverse results are generated to account for the higher number. We found 3 suggestions to leave enough room for predicting useful results with ambiguous input without generating messy results.

When testing for a placement match for an input query, we count a match if any of the 3 suggestions results in a match. If an exact match is encountered, it will be counted only as such, and no count for a close match is given for any other suggestion.

To counter the effect of possible invalid results, we run up to 100 trials per input and take the first 3 valid placements. The larger data sets contain

Artificial web	kNN	Transformer	GNN
Retrieval			
Valid	37.6 %	31.1 %	24.8 %
Overlap	57.6 %	42.0 %	64.8 %
Outside	4.8 %	26.8 %	10.4 %
Alignment	.000 [.000, .004]	.032 [.007, .092]	.035 [.018, .072]
Exact match	99.9 %	45.4 %	0.3 %
Close match	0.0 %	33.7 %	20.2 %
Failure	0.1 %	20.9 %	79.5 %
Generalization			
Valid	36.0 %	27.5 %	26.5 %
Overlap	58.6 %	44.2 %	64.4 %
Outside	5.4 %	28.3 %	9.1 %
Alignment	.000 [.000, .004]	.037 [.007, .092]	.039 [.021, .078]
Exact match	97.7 %	31.9 %	0.0 %
Close match	0.8 %	45.2 %	20.6 %
Failure	1.5 %	22.9 %	79.4 %

Table 6.3: Evaluation statistics for the 'artificial web' data set. Values in brackets correspond to the 0.25 and 0.75 quantile.

a vast number of queries (as noted in Table 6.1), and testing every possibility is computationally expensive. Hence, we limit the number of test queries to 1,000 in each of the retrieval and generalization cases, while preserving the distribution of element types, and considering all areas of placements on the canvas equally.

Tables 6.2, 6.3, 6.4, and 6.5 show the quantitative results of our evaluation. The alignment scores are displayed as *median [0.25 quantile, 0.75 quantile]*.

The kNN method achieves a very high retrieval match score on all data sets with scores of more than 95 %, and also the best generalization match scores in the handcrafted data sets between 70 % and 97 %. Matches are mostly 'exact', only in the first data set a significant share of generalization queries are matched only closely (14 %) which is otherwise around 1 %. The rate of valid results decreases as the layouts become more complex, from around 70 % in the simplest 'varying buttons' data, to $\tilde{2}5$ % in the Enrico data set that has the most elements, and the rate is similar for both retrieval and generalization queries. Invalid results are mainly due to overlap (between 28 % and 58 %), placements outside the canvas vary between 5 % and 25 %. The alignment scores are lowest for all data sets compared to the other

Enrico	kNN	Transformer	GNN
Retrieval			
Valid	25.8 %	35.6 %	19.5 %
Overlap	53.3 %	44.7 %	76.3 %
Outside	20.9 %	19.7 %	4.2 %
Alignment	.056 [.007, .098]	.096 [.048, .192]	.067 [.036, .126]
Exact match	98.4 %	8.3 %	0.0 %
Close match	1.0 %	29.5 %	9.0 %
Failure	0.6 %	62.2 %	91.0 %
Generalization			
Valid	23.9 %	34.1 %	25.4 %
Overlap	55.4 %	38.0 %	68.3 %
Outside	20.7 %	27.9 %	6.3 %
Alignment	.084 [.047, .129]	.112 [.064, .224]	.104 [.064, .186]

Table 6.4: Evaluation statistics for the 'Enrico' data set. Values in brackets correspond to the 0.25 and 0.75 quantile.

methods with a median of below 0.1 in all cases.

The Transformer model generates valid results in 30-50 % of cases. The rate of overlapping elements ranges between 26 % and 45 %, while placements outside the canvas occur in 20-30 % of cases. It achieves high retrieval matching scores of 70-80 % of cases except for the Enrico data set, where it is below 30 %. The generalization accuracy is similarly around 80 %. However, the rate of exact matches accounts for only 25-75 % of matches, and often the majority are only close matches. The alignment scores are three times the highest of the methods and surpass 0.1 in two cases. For the handcrafted data sets they are similar between the retrieval and generalization queries, while for the mobile layouts Enrico and Rico, there is an increase visible between the conditions.

The GNN approach produces only 20-50 % valid results on the different data sets and often produces overlapping placements. While for the smallest and simplest data set 'varying buttons' it closely matches patterns with decent rates of 63 % in the retrieval case and 50 % in the generalization case, in the more complex and larger data sets, it drops to less than 10-20% and around 90 % unmatched patterns. There are hardly any exact matches, only around 1 %. The alignment scores have a wide range and are very close to the kNN method on the simple data sets but increase to the highest score for the 'Rico NDN' data set where the median is 14-40 % above the alignment score of the other methods.

Rico NDN	kNN	Transformer	GNN
Retrieval			
Valid	34.6 %	53.2 %	27.9 %
Overlap	42.6 %	26.8 %	66.9 %
Outside	22.8 %	20.0 %	5.2 %
Alignment	.087 [.047, .141]	.096 [.064, .176]	.123 [.079, .201]
Exact match	97.8 %	19.2 %	1.3 %
Close match	1.0 %	49.8 %	13.6 %
Failure	1.2 %	31.0 %	85.1 %
Generalization			
Valid	36.0 %	53.5 %	32.4 %
Overlap	39.4 %	26.0 %	64.0 %
Outside	24.6 %	20.5 %	3.6 %
Alignment	.092 [.046, .170]	.112 [.064, .192]	.128 [.079, .200]

Table 6.5: Evaluation statistics for the 'Rico (NDN)' data set. Values in brackets correspond to the 0.25 and 0.75 quantile.

Next, we present a few example queries and the corresponding placement suggestions of the methods.

6.5 Qualitative evaluation

To better understand the real capabilities of each method, this section shows example results for the different data sets and contrasts it with expectations. For each data set, a handful of *retrieval* queries are evaluated, as well as *generalization* queries. For the GNN method, we show invalid results as there would often be no suitable output available. For the other methods, we only show valid placements.

Varying buttons. This data set contains similar placements for most elements except for buttons that can have multiple valid locations. If two elements of the same types are present, they should be located next to each other. Figure 6.9 shows three example retrieval queries (i.e. it was part of the training data). As such, we expect to see valid and matching results if it was learned properly.

The Figures 6.10, 6.11, and 6.12 show the results for the queries *a), b), c)* with the different models.

For query *a)* with a static position of the target element, all models produce similar results. Still, the transformer and kNN methods suggest as

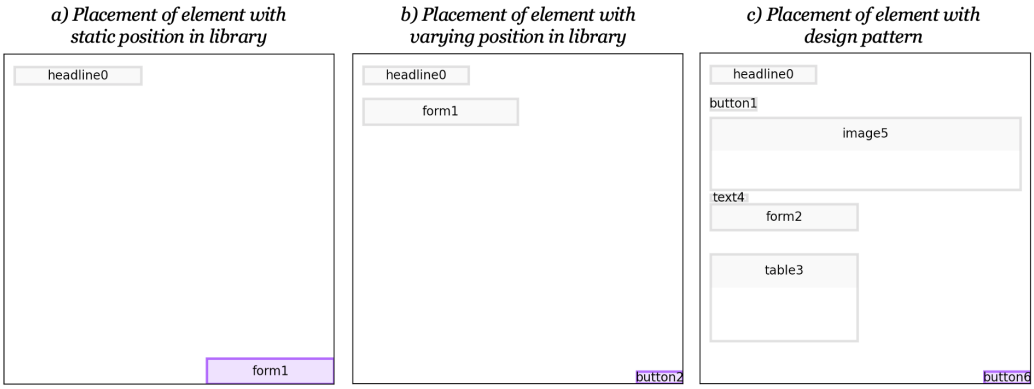

Figure 6.9: Retrieval queries for the *varying buttons* data set.

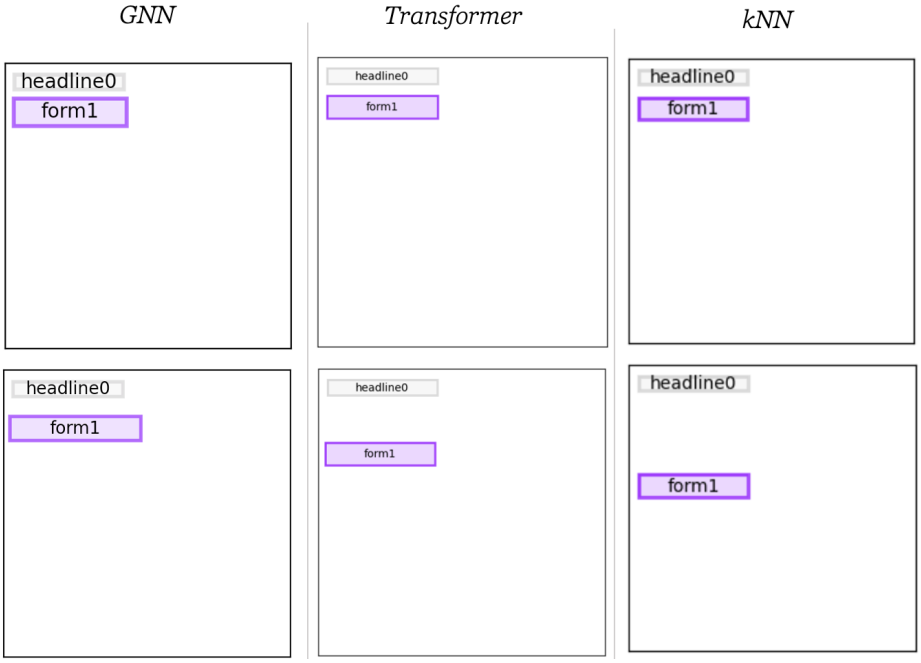

Figure 6.10: Results for the retrieval query *a)* with *varying buttons*.

an alternative a much lower placement which is seen when an image element is present above.

For query *b)* with a varying position in the training set of the target element, the results are quite different. The GNN method tries to squeeze the button between headline and form which also exists in a similar composition (although then there is more space between those elements) and produces overlapping placements. The transformer model suggests a left-

Figure 6.11: Results for the retrieval query *b)* with *varying buttons*.

aligned placement and a placement on the right side, which is, however, not following the pattern of being below the form. The kNN model suggests two placements according to neighbors, at the right side of the canvas and below the form, which matches different training elements.

For query *c)*, where the placement follows a layout pattern (next to the element of the same type), we see similar results as in the previous query. The GNN method again tries to squeeze the button between headline and image and produces overlapping placements. The transformer model suggests a correct placement next to the other button and a placement on the right side in the same row which is creative. The kNN model suggests two similar placements, once also next to the other button, and once below the form on the right side. Since there is only one really valid position in this case, the secondary suggestions of the transformer and the kNN model can be seen as explorative ideas.

Figure 6.13 shows three sample generalization queries (i.e. it was **not** part of the training data). Hence, the results can be more different between the models and indicate the generalization capabilities.

The figures 6.14, 6.15, and 6.16 show the results for the generalization queries *a), b), c)* with the different models.

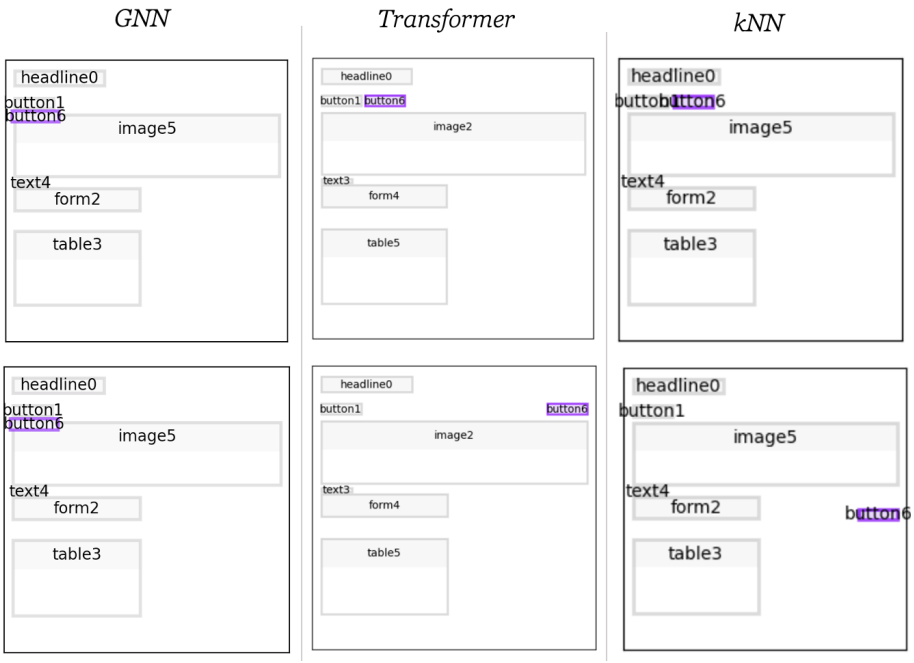

Figure 6.12: Results for the retrieval query *c)* with *varying buttons*.

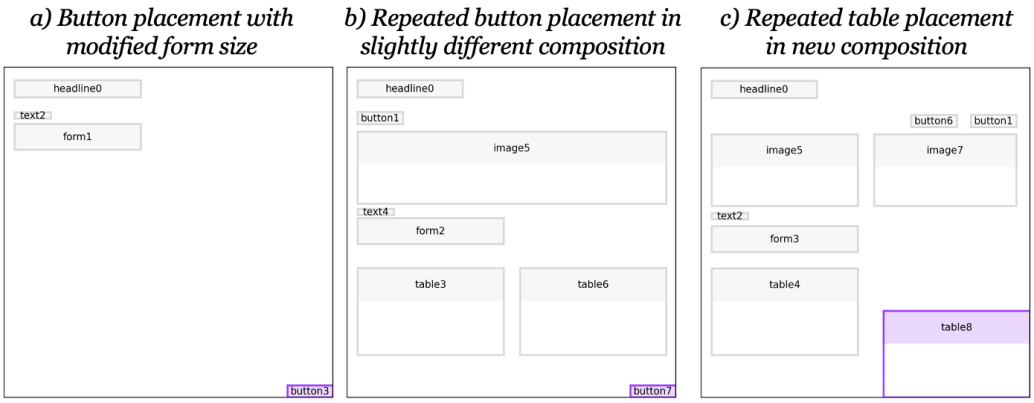

Figure 6.13: Generalization queries for the *varying buttons* data set.

For query *a)*, the GNN produces overlapping elements that are in a similar position as previous button results. The transformer model predicts two valid positions that correspond to encoded patterns of the data set. The kNN method also predicts two valid positions that are following encoded patterns (as there are three button positions for bottom placements).

For query *b)*, the GNN generates overlapping placements in a similar position as for the other button queries. The transformer model finds two valid

Figure 6.14: Results for the generalization query *a)* with *varying buttons*.

results but the top placement is incompatible with the overall layout system. The second placement is similar to other patterns but not the expectation in this case. The kNN model returns the expected position at the first result where the button is placed next to the other button. The second placement is a creative result but not according to a real design pattern.

For query *c)*, there is only really a single placement possible and valid. As all methods converge to the same result, only a single variation is shown. The GNN method puts the new table at the position of the existing table, resulting in an invalid overlapping placement. The transformer model and the kNN method return the same position at the expected position.

Artificial web layouts. This data set encodes three layout patterns (Table + Button, Form arrangement, and Dashboard layout) which are present in many different contexts. Figure 6.17 shows three example retrieval queries with these patterns. As before, we expect to see valid and matching results if it was learned properly.

The figures 6.18, 6.19, and 6.20 show the results for the queries *a), b), c)* with the different models.

For query *a)* where a button is to be placed and a table is already present

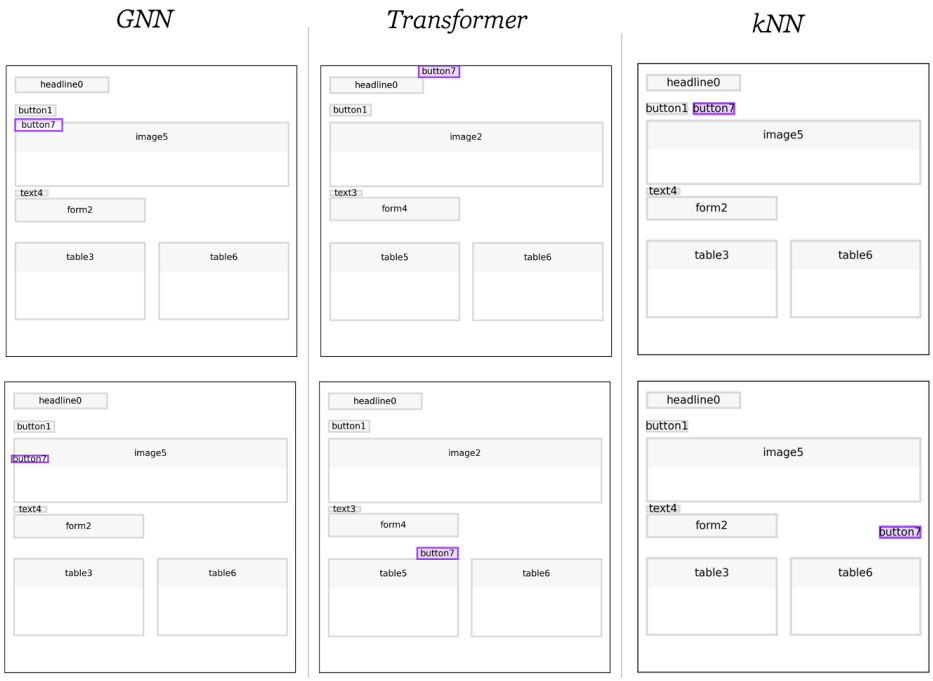

Figure 6.15: Results for the generalization query *b)* with *varying buttons*.

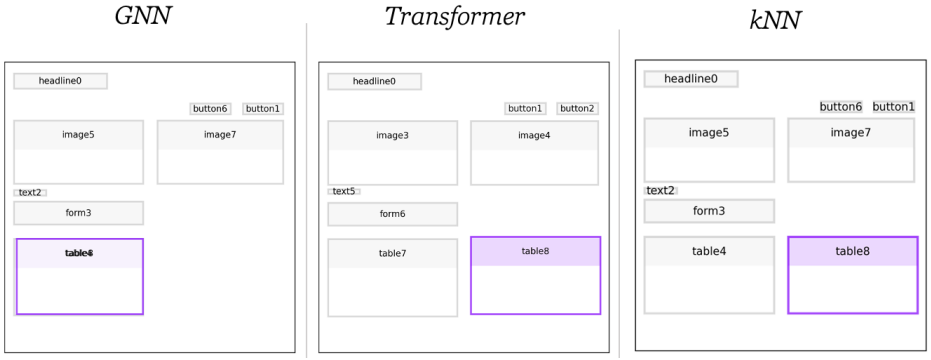

Figure 6.16: Results for the generalization query *c)* with *varying buttons*. All methods converge to the same output so only one variation is shown.

on the page, we would expect to see that it is placed above or below it, always right-aligned. Buttons are also present in other positions in the data set so it is not trivial. The GNN model does not return expected placements, instead, it centers the button similarly to the button that is already on the page. The variations are not greatly different. The Transformer model predicts one expected position, putting the button above and to the right of the table.

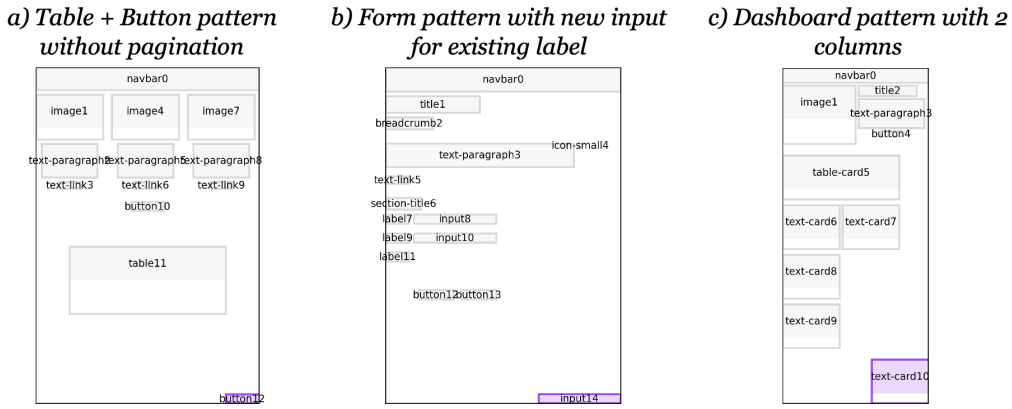

Figure 6.17: Retrieval queries for the *artificial web* data set.

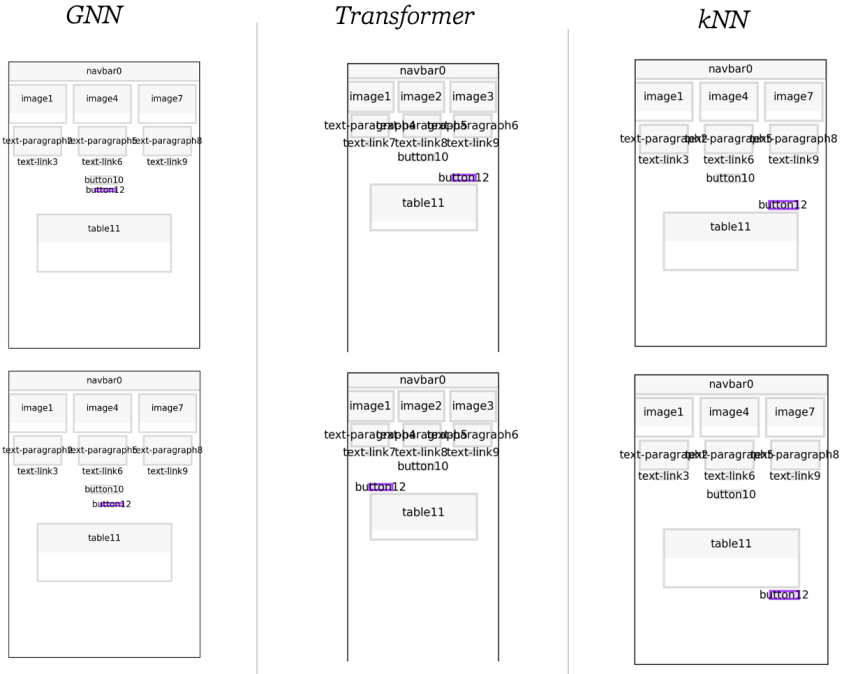

Figure 6.18: Results for the retrieval query *a)* with *artificial web*.

The other position is well-aligned, but not following the patterns of the data set. The kNN approach suggests both expected positions, placing the button above and below the table, on the right side.

For query *b)* where a small form is already in the layout, one row is with a label but without an input and a new input should be added. We expect it to be placed next to the label so it aligns well with the other form rows. The

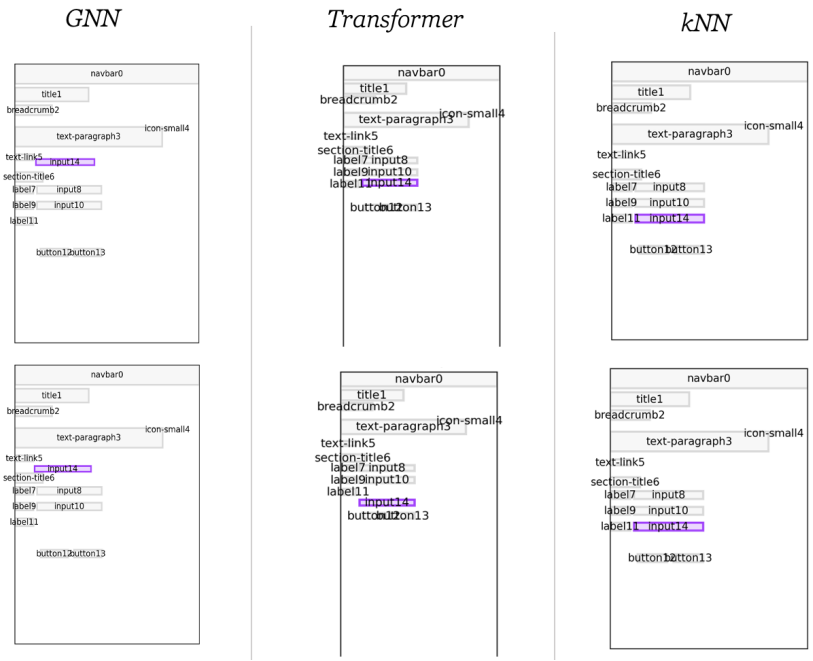

Figure 6.19: Results for the retrieval query *b)* with *artificial web*.

Figure 6.20: Results for the retrieval query *c)* with *artificial web*.

GNN model places the input above the form in both variations, and although rather aligned, it is not a sensible result in this context. The Transformer suggests the expected placement next to the label and another one in a new row. The kNN method only suggests the expected position.

In query *c)*, a new card is to be placed in a dashboard layout that has two columns. There are two positions that would make sense: in the first column below the other cards, or in the second column making the layout more balanced. The GNN method fails to predict any of these two expectations. It even shrinks the element but places it in the corridor of the second column. The Transformer model does not produce valid results for this query, the placements are overlapping or outside the canvas. This indicates that it did not learn to differentiate between the different column layouts of the dashboard. The kNN approach suggests the two expected positions.

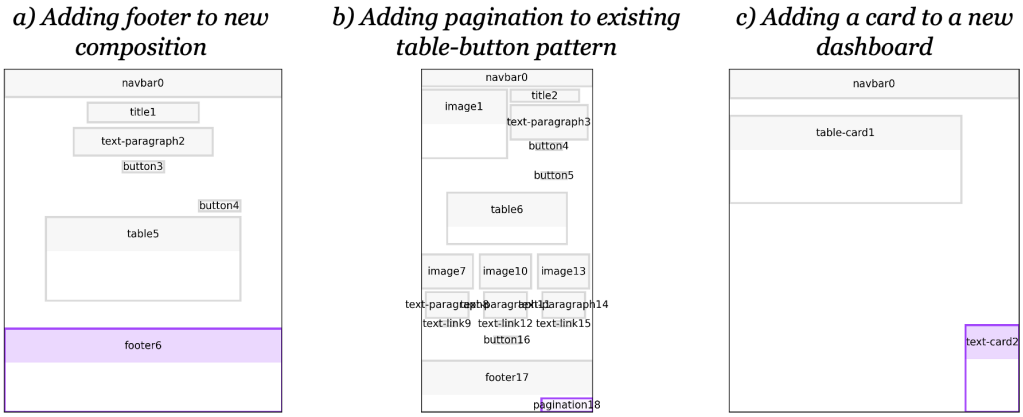

Figure 6.21: Generalization queries for the *artificial web* data set.

Figure 6.21 shows three generalization queries that are not part of the training set.

The figures 6.22, 6.23, and 6.24 show the results for the generalization queries *a), b), c)* with the different models.

In query *a)*, a footer is to be placed to a completed layout. The natural position would be at the bottom of the canvas. While this is always the case in the dataset, the actual coordinate varies depending on the height of the layout. As such, the GNN method predicts two positions that overlap with the existing table element and also shrinks the size of the element. The Transformer model surprisingly fails to place this element correctly and generates positions that overlap with the other elements, or that are outside of the canvas. The kNN method places the footer correctly at the bottom of the canvas with different gaps to the previous element.

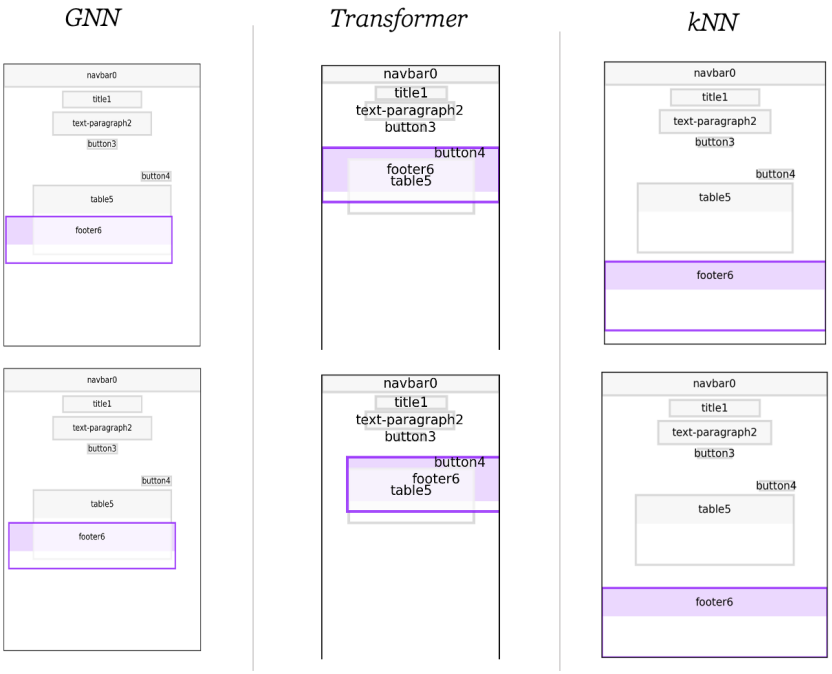

Figure 6.22: Results for the generalization query *a)* with *artificial web*.

For query *b)* where a pagination element is to be placed to a complete layout, we expect that it follows the layout pattern where the pagination is always around the table, in the same row as the button that is above or below the table. The GNN method produces two positions that overlap either with the top header elements or the table and thus are not usable as such. The transformer model predicts positions that are close to the expectation but are overlapping with the table. This indicates that it was not able to transfer the pattern to a new area on the page that was not part of the training data. This is what we call the problem of a shifted pattern. The kNN model returns the positions as expected where the pagination is above the table and next to the button.

For query *c)* where a new card is to be added to a rather empty dashboard layout, we expect that it is placed in the second row, and ideally at the start of the row. The GNN method fails to predict a valid result and shrinks the element unexpectedly. The transformer model returns two valid positions that both are in line with our expectation, where one is also placed at the start of the row. The kNN model suggests placements in the new row but does not return a placement at the start of the row.

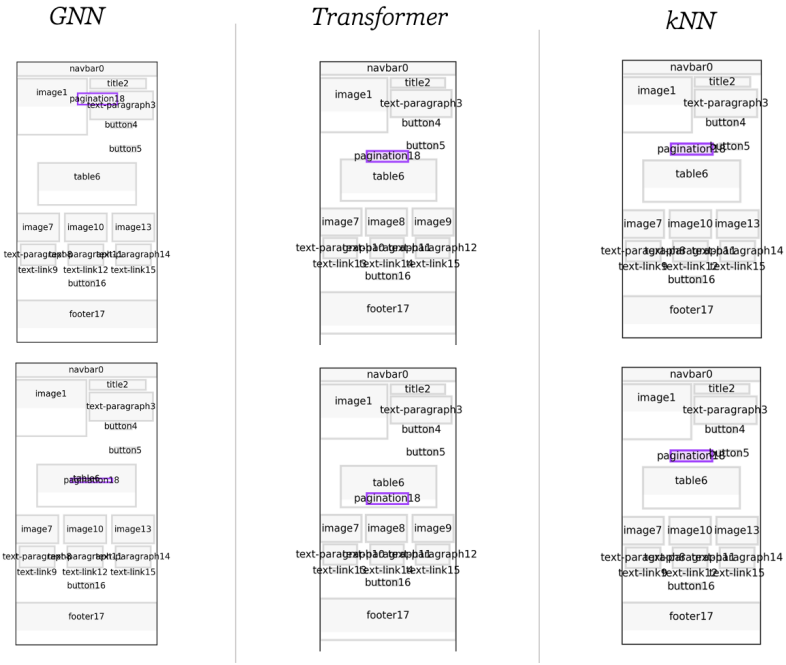

Figure 6.23: Results for the generalization query *b)* with *artificial web*.

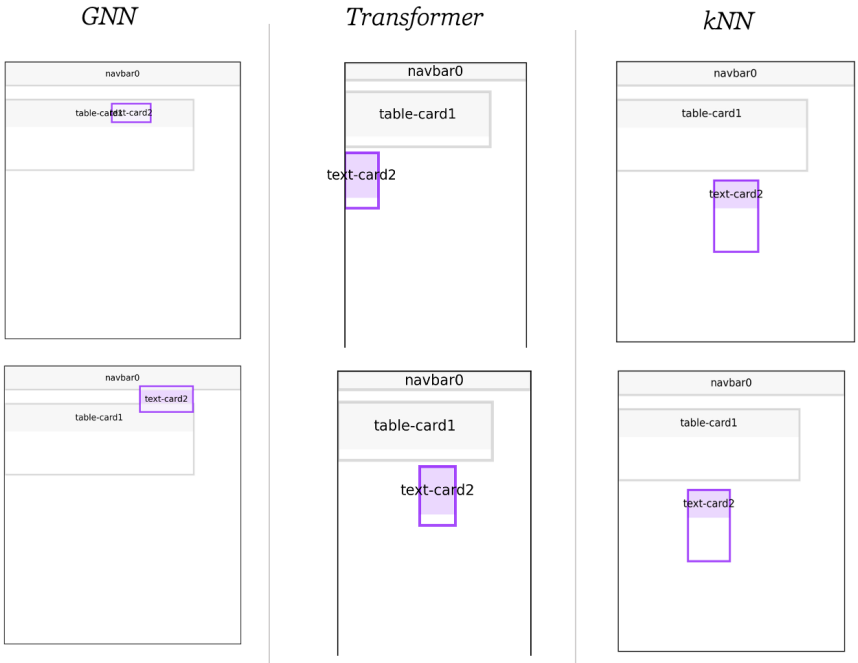

Figure 6.24: Results for the generalization query *c)* with *artificial web*.

Enrico. The mobile layouts based on Enrico's data set features mobile layouts with element counts ranging from 2 - 38. As there are no known layout patterns encoded, we can only evaluate if the generated placements are generally producing well-formed layouts and are valid. Figure 6.25 shows three example retrieval queries. As before, we expect to see valid and well-formed results if it was learned properly.

Figure 6.25: Retrieval queries for the *Enrico* data set.

The figures 6.26, 6.27, and 6.28 show the results for the queries *a), b), c)* with the different models.

In query *a)*, a new input element is to be added where a single centered image element is already present on the page. The GNN method predicts two invalid and overlapping placements. The Transformer method suggests two placements that are above and below the image and that appear centered, so they can be considered well-formed results. The kNN approach returns two placements below the image with different offsets that are center-aligned with the image, so they can be considered well-formed results as well.

In query *b)*, a set of list items is present in the layout and another one should be added. The GNN again fails to present valid non-overlapping results. The Transformer method suggests placements towards the top of the screen that results in overlaps with existing elements. The kNN approach returns two placements below the existing list items, and even though the gaps with the previous list item is not the same as with the other list items, we consider it well-formed suggestions.

In query *c)*, a more complex layout is already present and a new text button element should be placed. There is a specific gap at the end of the page where it would fit in where it is also present in the training set. The GNN approach generates two placements in the middle of the screen

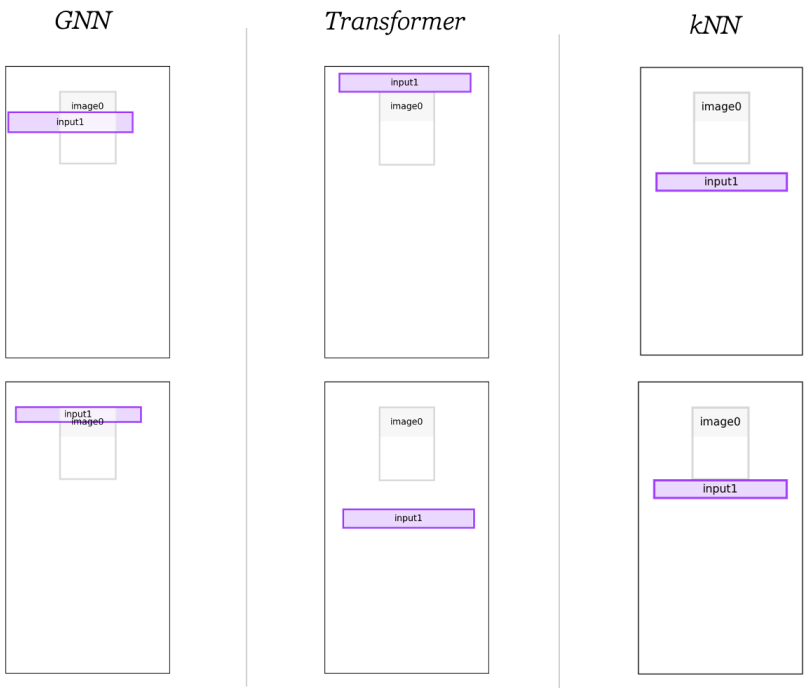

Figure 6.26: Results for the retrieval query *a)* with *Enrico*.

that appear nearly aligned with other elements. While not returning our expectation, it can be considered a good result. The Transformer method predicts one placement at the top of the screen and another one in the middle of the screen, and both are right-aligned to other elements. While also not following the training sample, it can be considered a good result. The kNN model suggests the expected position and a second one in the middle of the screen, providing the most useful results.

Figure 6.29 shows three generalization queries, and the figures 6.30, 6.31, and 6.32 show the results with the different models.

In query *a)*, two elements already occupy a large portion of the screen, and a new text element is to be added. The GNN method predicts two invalid placements that overlap with the existing image element. The Transformer model predicts similarly invalid placements that overlap with the image element. The kNN model returns two valid placements below the existing elements, and the first result is well-aligned with both elements.

In query *b)*, a small image element should be added. The top of the screen is occupied by existing elements and at the bottom are two other small images and icons. The GNN method predicts one valid result where the image is centered below the list items and one invalid result where the

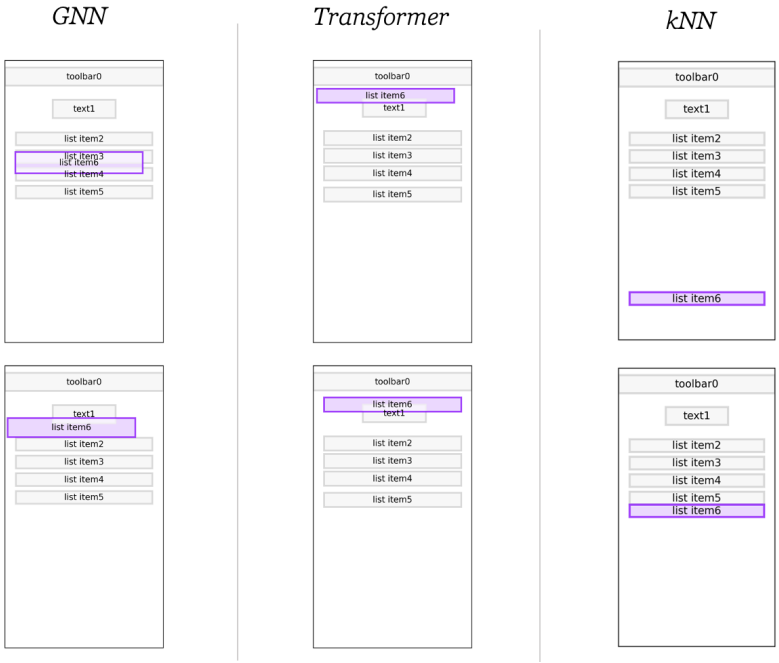

Figure 6.27: Results for the retrieval query *b)* with *Enrico*.

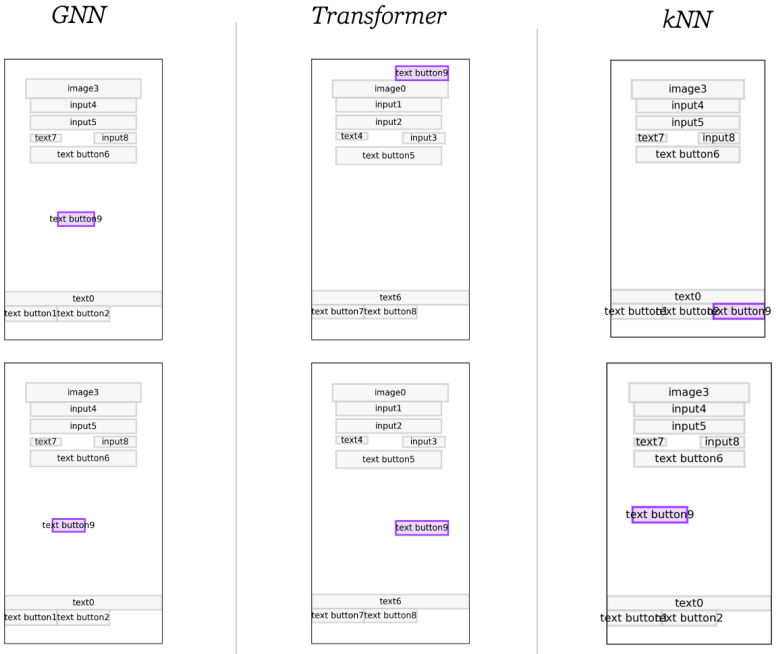

Figure 6.28: Results for the retrieval query *c)* with *Enrico*.

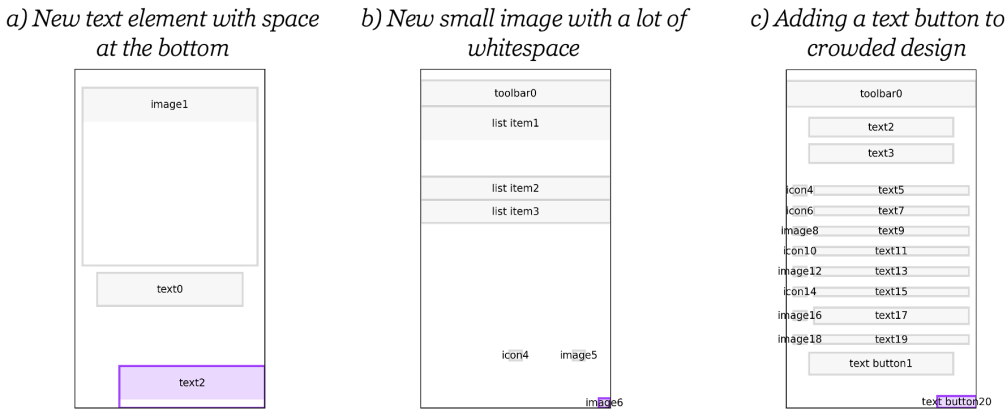

Figure 6.29: Generalization queries for the *Enrico* data set.

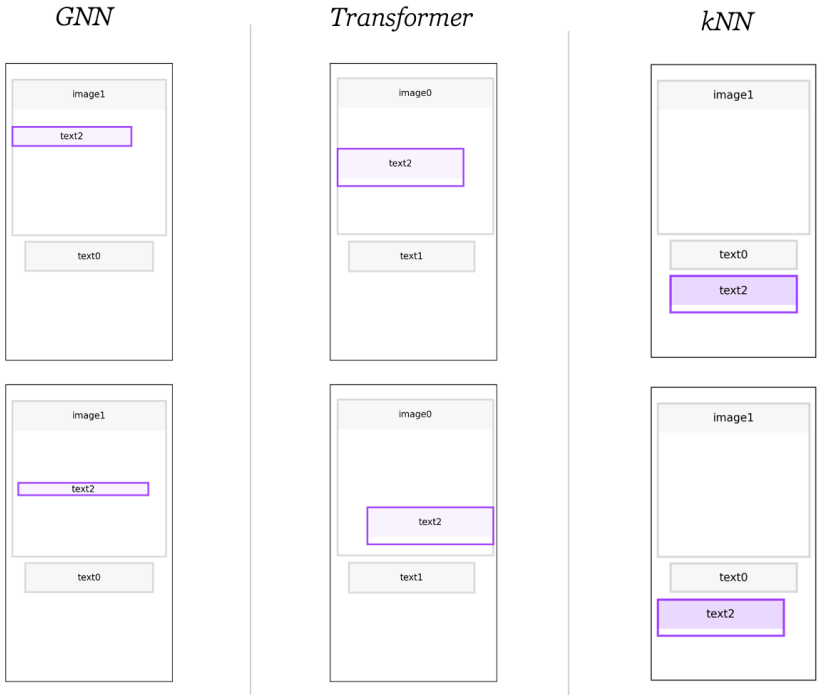

Figure 6.30: Results for the generalization query *a)* with *Enrico*.

image is overlapping with the existing elements. The transformer model shows two results where the image element is at the edge of the screen, and both can be considered good suggestions. The kNN model also predicts two valid placements, one where it is well-spaced with the other small images at the bottom, and one where it is placed between the two small images. Both can be considered good suggestions.

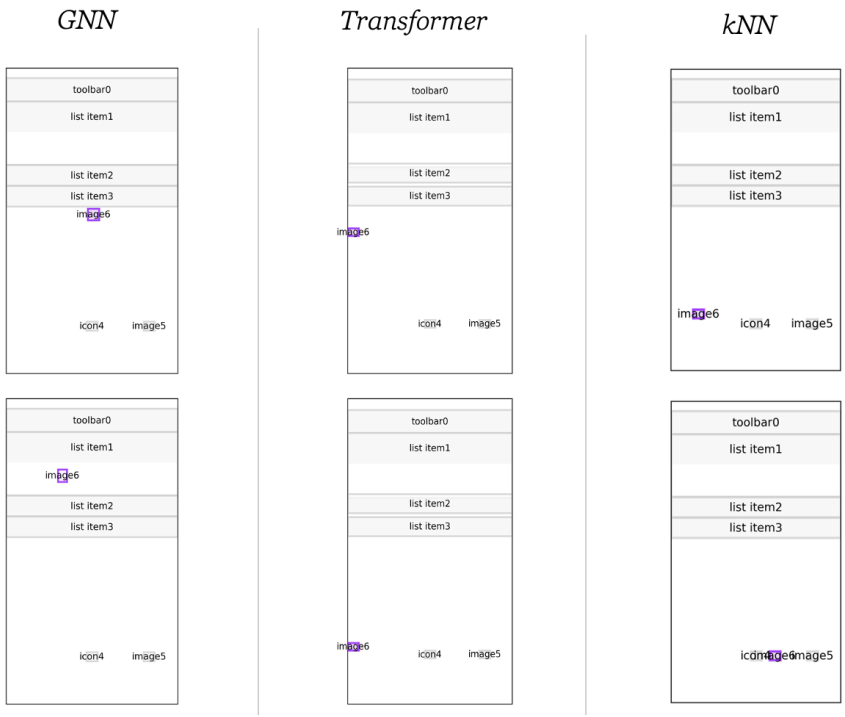

Figure 6.31: Results for the generalization query *b)* with *Enrico*.

In query *c)*, a more difficult case is evaluated where the page already contains 20 elements and a new small text button is to be placed. There is not a lot of room available for that though. The GNN method predicts the same placement where the element is added between the existing text elements, although no clear alignment is visible. The transformer model returns two invalid results that overlap with existing elements on the page. The kNN model returns similar placements to the GNN model and places the element between existing text elements with enough space in between. The first result exhibits a center-alignment with another text element and can be considered a good suggestion.

Rico (NDN). In this data set, the models have a much larger set of training items available where all layouts have less than 10 elements per page. Figure 6.33 shows three example retrieval queries. As before, we expect to see valid results if it was learned properly. The figures 6.34, 6.35, and 6.36 show the results with the different models.

In query *a)*, the screen contains 5 elements spanning the majority of the upper area and a new text button should be added. The GNN model returns valid positions and places the text button above the image at the top area

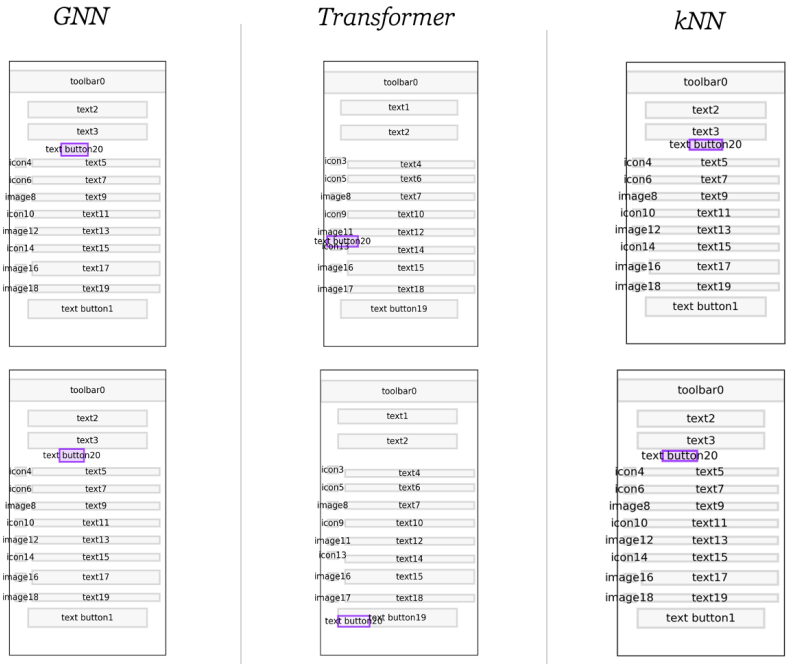

Figure 6.32: Results for the generalization query *c)* with *Enrico*.

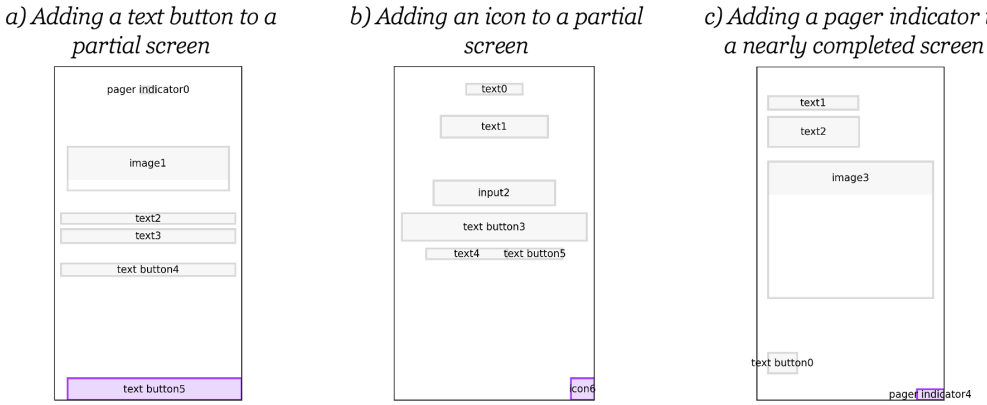

Figure 6.33: Retrieval queries for the *Rico (NDN)* data set.

of the screen. The Transformer model predicts placements where the new element is below the existing elements with different gaps to the existing elements. The kNN model returns similar results but both positions are more towards the bottom of the page. Interestingly, in this case, all models returned valid and well-formed but not diverse results.

In query *b)*, six elements located at the upper half of the screen are present

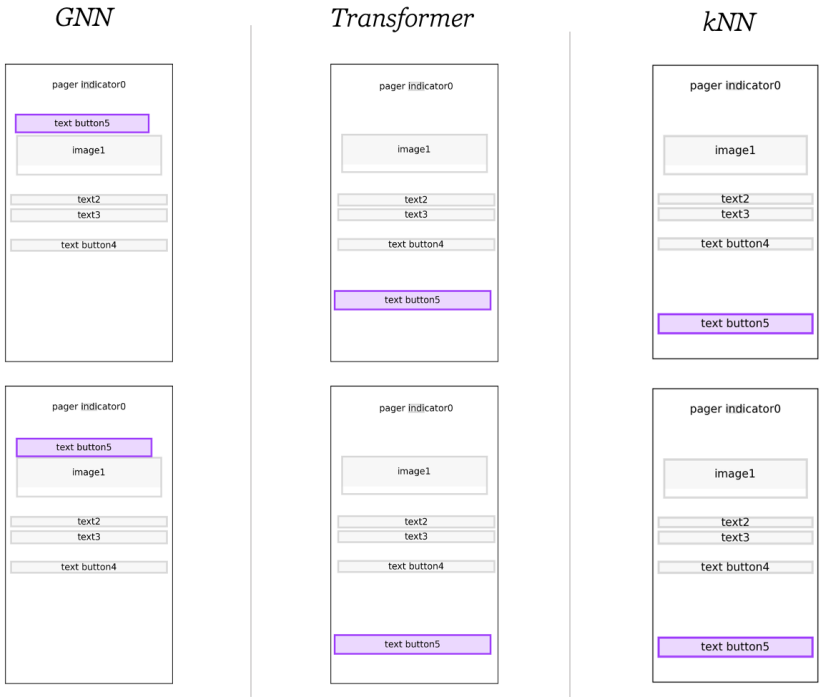

Figure 6.34: Results for the retrieval query *a)* for *Rico (NDN)*.

and a new icon should be added. The GNN model predicts placements where the icon is added above or below the second text element but without alignment to other elements. The Transformer network places the icon at the left edge of the screen, in the same row as the text element. Both results are aligned to either the text element of the same row or the text element in the middle of the screen. THe kNN model predicts similar placements, at the top left area of the screen in the row of the first text element. The second suggestion is closer to the middle though and not aligned.

In query *c)*, four elements that span the whole screen already exist in the layout and a pager indicator element is to be added. The GNN approach places the pager indicator between the text and image elements without alignment. The Transformer model predicts results where the new element is towards the bottom of the screen and horizontally centered, but with two different y-coordinates. The first result is aligned with the bottom text button, while the second result is only center aligned with the screen. The kNN method places the pager indicator also either centered at the bottom of the screen, or at the top, and left-aligned. Both results can be considered good suggestions, thus.

Figure 6.37 shows three generalization queries, and figures 6.38, 6.39, and

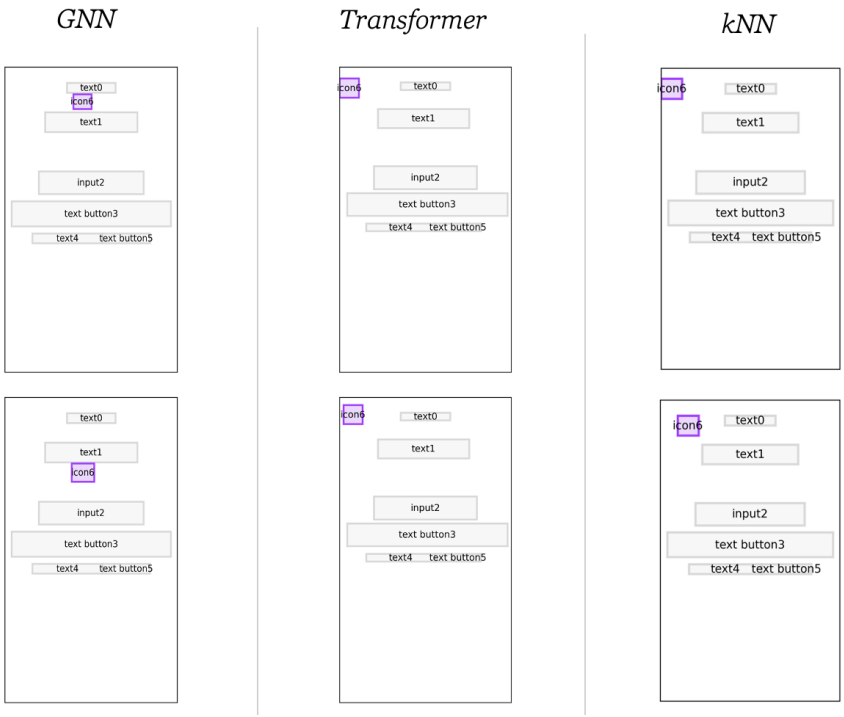

Figure 6.35: Results for the retrieval query *b)* for *Rico (NDN)*.

6.40 show the corresponding results with the different models.

In query *a)*, a toolbar is to be placed with three elements being already present on the screen. In any case, we would expect the toolbar to be positioned at the top of the screen. Since the placement of a toolbar is so unambiguous, all models return a placement at the top of the screen without variation, hence, only a single row of results is shown. The only notable difference is that the GNN method shrinks the element's width making it not span the full width anymore. The Transformer and kNN place it as expected at the top from left to right.

In query *b)*, six elements fill the middle portion of the screen and a new text element should be added. The GNN method returns two invalid results where the element overlaps with the existing layout. The transformer model predicts placements at two very distinct positions (top and bottom). While the top placement appears less likely, without context it must be considered well-formed together with the bottom placements, as both are full-width and thus, well-aligned. The kNN model positions the text element at two areas at the bottom of the screen and that also span the full width and are well-aligned and can be considered good suggestions.

In query *c)*, five image elements are placed at the top of the screen and

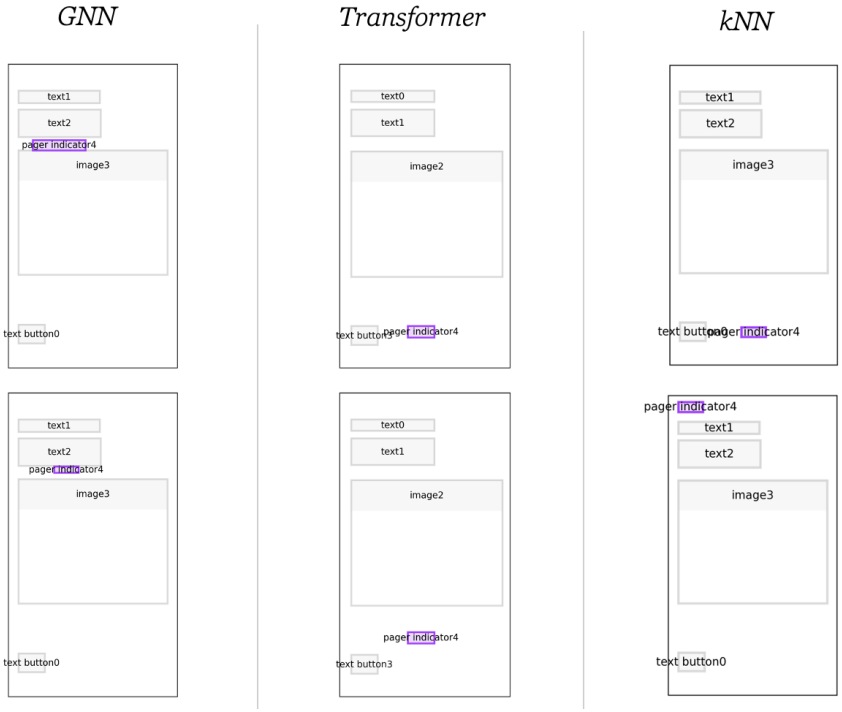

Figure 6.36: Results for the retrieval query *c)* for *Rico (NDN)*.

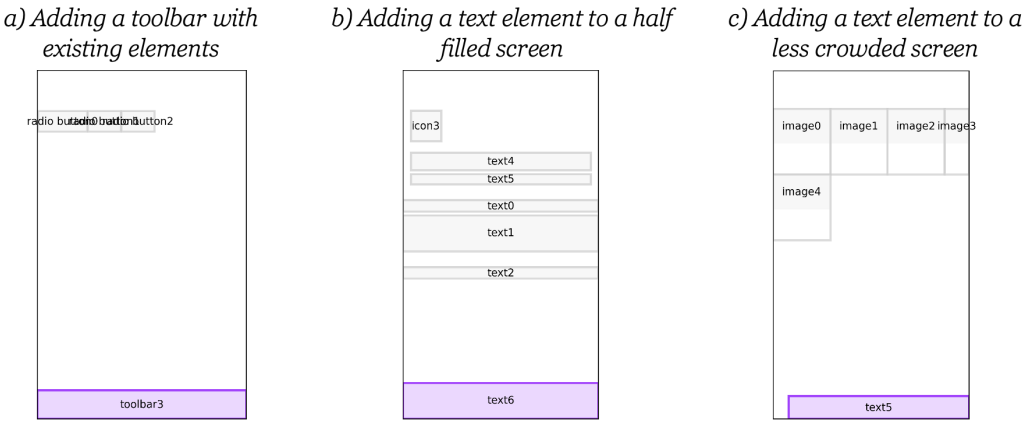

Figure 6.37: Generalization queries for the *Rico (NDN)* data set.

a new text element is to be added. The GNN method returns again two invalid positions that overlap with other elements. The transformer model predicts placements at the top of the screen, above the images, where one result is centered, and the other left-aligned. The kNN model shows similar results and places the text element above the images but keeps it left-aligned

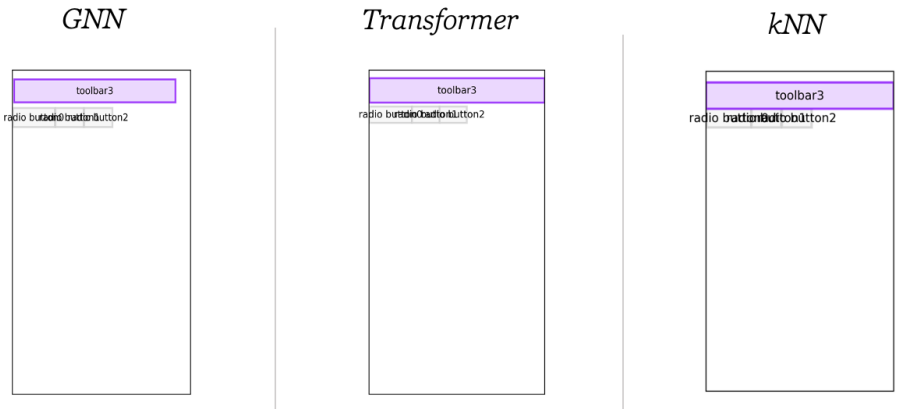

Figure 6.38: Results for the generalization query *a)* for *Rico (NDN)*.

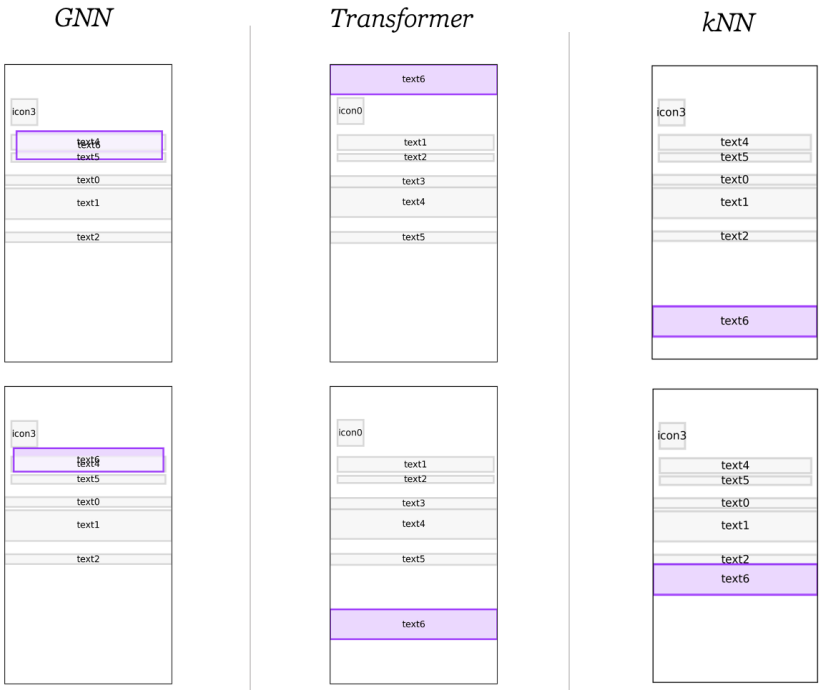

Figure 6.39: Results for the generalization query *b)* for *Rico (NDN)*.

in both cases. Both the Transformer and the kNN results can be considered good suggestions.

Summary We have generated different suggestions with the three models for 6 cases per data set and inspected the results. We can notice that

Figure 6.40: Results for the generalization query *c)* for *Rico (NDN)*.

the GNN mostly does not produce very diverse results and predicts many overlapping placements. The Transformer produces mostly valid and well-formed results but fails in a few cases as well. The kNN method is able to suggest valid results in all inspected queries and is more consistent than the Transformer.

Next, we will discuss these findings in the upcoming chapter.

Chapter 7

Discussion

In this work, we have evaluated three different methods on the stated element prediction problem: two recently proposed neural network approaches based on a GNN and on the Transformer architecture, as well as our own method that employs sequence alignment algorithms and nearest neighbor search to produce placements.

To give designers control over a possible application, we condition the input on the user-defined element type and size which should be added to an existing layout. We focused on pattern matching when placing new elements next to general layout qualities as layout patterns play an important role in UI design.

Graph neural network We found that the implemented graph neural network following the description in 'Neural Design Network' [21] achieves the lowest scores among the tested methods across all data sets in nearly all tested metrics. Especially problematic is the high number of invalid results due to overlapping with other elements that require a large number of samples to generate usable results. Even so, it has the lowest pattern matching scores, indicating that it is not learning the patterns as expected. Further, it does not respect the given sizes of the elements in many cases resulting in unexpected outcomes. While the graph representation is a natural and useful format for layouts, our implementation is not able to produce high-quality results most of the time.

We also tried using only the relation module to produce constraints for a combinatorial optimization system. However, the predicted edges were often not consistent with each other such that no feasible result was possible. These conflicting relations might also be a reason for the layout module to produce overlapping placements. In a small test, where a good and consistent set of relations is given directly to the layout module, the results improved

significantly.

In our evaluation, the ability to produce diverse results was limited, and the model seemed to have converged to a small region even for different latent codes. While high creativity is not required, there were often multiple valid placements, which were not returned by the network.

Since there are many parameters and details in this method that are not fully described in the original paper by Lee *et al.* [21], it cannot be excluded that differences or errors in the implementation are responsible for these low scores. Further, our restrictions on the problem, requiring non-overlapping results, and performing single element predictions might require changes to the model. While we tested different variations to improve the results, we did not achieve significantly better scores with any of the modifications.

Transformer The transformer model implemented similar to the proposal by Gupta *et al.* [11] produced overall good results. It was able to produce close results of the majority of the patterns in the handcrafted data sets and produced well-aligned results in many cases.

Reducing the canvas size by employing a base grid is a valuable approach that limits the necessary prediction categories and simplifies the decision of the placement. In addition, modeling the coordinates as categories instead of a continuous variable, allows the model to learn the alignment lines of the layout set such that it is more likely to generate similar position values on the dominant alignment axes.

That has the disadvantage, however, that vertically shifted patterns due to additional elements will not result in equally shifted predictions if this position value was not found in the training set. Qualitative results support this argument.

While the results are reasonably well even for our small data set, the results are not good enough to be fully usable in many cases. Since the complexity of the smallest data set is very low, we can conclude that it is not suitable for small data sets.

The ability to define the temperature parameter when decoding the next tokens allows users to balance the need for "safe" and valid results versus "creative" results. While generally, creative exploration is not the target of this work, the possibility of that feature is a useful enhancement. Our tests showed that results with a low temperature ($t = 0.1$), and consequently conservative prediction, increases the rate of valid results significantly, however, also limits the diversity of prediction. Increasing the temperature moderately ($t = 0.2$) where it does not change the probability distribution of the Transformer network significantly, increases the variety of results at the cost

of a higher rate of invalid results. Nevertheless, the matching accuracies were not affected by this change significantly. Hence, we suggest that for practical use, a low temperature is most appropriate.

The qualitative results show that for new compositions, many predictions are invalid, especially if the layout is already crowded and only a particular empty space would be available. Since there is no explicit understanding of the visual nature of the layout, these layout holes are not recognized. Extending it with convolutional layers that read in the visual layout might be able to resolve this problem.

Finally, label smoothing is applied on the full set of tokens, across all ranges for the different element attributes. Restricting the label smoothing on the valid range of the current element position might further improve results and strengthen awareness of the attribute position in the sequence.

Sequence-aligned nearest neighbor search To address these limitations, we proposed our own method that employs sequence alignment algorithms and explicit layout quality principles for generating features for a nearest neighbor search algorithm.

It was able to predict layout patterns with the best scores among the other methods, as it explicitly leverages the reference designs that contain these patterns and searches for fitting neighbors. However, as it does not learn generalizations of these patterns and takes a single neighbor as the reference, it is not able to use information from multiple designs to inform the prediction. Certainly, one possible extension is to take into account multiple neighbors for a single input query, however, since the placement is based on the neighborhood relations of the returned neighbor, multiple neighbors might return conflicting relation categories. While it is not trivial to extend it in that direction, it is a promising direction of future research.

Incorporating concepts from the graph representation of a layout during the placement step helps to circumvent issues from shifted patterns as the neighborhood relations are primarily used to produce the result. This, in turn, can lead to failure cases when the neighborhood is sparsely populated, or the relevant reference element for placement is not in the direct vicinity.

The runtime is a disadvantage, as it scales with an increase in element sizes and size of the reference designs since they are scanned for every input for the patterns. The biggest bottleneck in this regard is not the nearest neighbor algorithm which is very performant thanks to the usage of ball trees, but the actual enumeration of layouts from the set of references that are individually processed by the sequence alignment method beforehand. Intelligent filtering of promising candidates before the sequence alignment

could help to restrict the search and improve the runtime. Nevertheless, it is the inherent cost of instance-based learning algorithms that do not aggregate the data during a learning phase.

Another disadvantage is that its ability to produce diverse results is limited. The number of overlapping results or results outside of the canvas increases with the complexity of the layouts. Since the sequence alignment and feature calculation do not consider the final placement (as this is too costly to perform on all library items), there is a certain chance of generating neighbors that cannot be applied to the current input. Future research should investigate the possibility of incorporating a better approximation of the final placement into the feature generation process.

Lastly, the sequence alignment algorithm employed currently is further limiting its ability to produce valid results in certain cases because it is biased towards matching in the beginning of a sequence and does not allow to fine-tune the matching behavior. Improving this is a challenging task while keeping the performance of the algorithm. Another direction of improvement is to match rows of elements as subsequences separately in order to prevent matching elements across different rows.

However, a major advantage of this method is that it works well with small amounts of data which is more commonly found in commercial settings. Even large products might not exceed a few hundred designs. As such, it is important for the applicability of a method that it can produce good results with small data sets. The second major benefit of this method is that it is interpretable. Since the result is always based on a particular neighbor with a mapping between the elements of the two layouts, the results can be understood and errorneous outcomes can trigger improvements of the algorithm.

Layout representations Both layout representations contain assumptions that may not be valid in all cases.

The challenges of the graph representation lie in balancing the number of relation categories with their expressiveness, and understanding implicit relations: While we have tried to cover the most common alignment relation types, there might be others that are more useful with other data sets. Further, defining relations to the canvas can be difficult as it can lead to detected relations that are implicit due to the layout being packed, e.g., an element at the bottom of a layout could be considered to be inherently 'at the bottom', making the case that this type of element has a special relationship with the canvas edge (like a footer), or it could be at this position because it is dependent on the elements above it that push it to the bottom (and would

be valid in any other vertical area if below said element).

Additionally, we only support flat layouts in this setting. The original proposal supported container elements with corresponding relation types, hence, extending this should be straight-forward. Nevertheless, incorporating the ideas from Li *et al.* [25] where trees are used to represent layouts, we could represent layouts as hierarchical graphs where container nodes can themselves represent a 'mini-canvas' with a separate graph of its child elements, and thus, limit the number of elements per graph.

Similarly, the sequential decomposition follows a simple reading order without taking into consideration nested columns or segments of the layout even if not represented with an element (e.g., as indicated with whitespace). This is a difficult problem with existing research to segment layouts from the visual representation of a layout [2]. It could be alleviated by requiring explicit groupings of segments, which might be a natural way of working in a design application, or by requiring pseudo-elements to indicate common areas. Then, the layout decomposition can create better sequences that take into account columns and segments, similar to the sequential representation by Li *et al.* [25].

Applicability Both neural network approaches are data-hungry methods and require finetuning to produce the best results. Even though the transformer model generates promising results for the larger data sets, the problems with shifted patterns make it difficult to apply in practical settings as layout patterns are often locally informed and not bound to the absolute position.

Further, the difficult interpretability of the methods is likely to frustrate users that are presented with unexpected suggestions. Advancements in combining probabilistic methods and neural networks could help inform the user of the model's confidence in the result, which is currently not present in the proposed methods. It is also non-trivial to split up a data set into a training and testing partition if the number of encoded patterns is sparse, i.e., few layouts contain the same patterns. This leads to problems with overfitting and poor generalization.

All methods currently do not support directly hierarchical elements or overlap between elements which is an important feature in modern layouts with background images or container cards. This would need to be addressed in future work.

Hence, considering the applicability to practical use in a professional environment, we find the kNN method to be best suited. While not without flaws, it can work with limited data, learns directly on the given examples,

incorporates layout principles in its feature generation and placement result, and can present the existing design used as a reference, making the results understandable. This could help users learn over time for which inputs it is not suitable and use it as an additional tool in their own process.

Evaluated methods We did not consider the class of Bayesian methods in our work. Previous work in this area showed promising results [3, 42]. Applying these ideas to the stated problem would be an interesting alternative approach.

Similarly, a large body of work has applied combinatorial methods to the layout problems, and combining Machine Learning with Combinatorial Optimization is another alternative. In this way, explicit layout rules and design system guidelines can be encoded for the placement step in the kNN method.

Evaluation Our main limitation in the evaluation is the lack of user feedback on the results. Conducting a study on the usefulness of the suggestions is highly desirable. However, the problem of having a consistent data set of layouts available persists which we encountered when evaluating the methods as well. As the context of the use case lies in the application to a single system, setting the context of the study is extremely important and can be challenging to achieve because participants will have to get familiar with the system's background and existing designs.

Comparing the evaluation of previous research to our own evaluation and results, we found that previous evaluations were conducted on data sets that do not allow explicitly verifying if the learned patterns correspond to actual layout patterns of design. We argue that this is an important element when evaluating methods for UI design. Certainly, there are general layout principles that indicate if a layout can be considered well-formed, such as alignment, but as layout patterns play an important role in UI design, they need to be evaluated in the context of UI layout generation and completion.

The final conclusion in the next chapter provides a summary of the work and provides pointers for future work.

Chapter 8

Conclusion

In this work, we have evaluated three methods for the layout completion problem with constraints. We focused on a practical application that is relevant for commercial designers working on a single system. In this context, the spatial consistency of element placements to the patterns found in existing designs of the same system plays an important factor. As such, we give designers control over the generation and let them decide the type of the next element as well as its dimension. The methods' goal is then to predict the position of this element on the partial layout, such that layout patterns from previous designs are followed. We measure these layout patterns in terms of relative position and alignment similarity of the neighborhood of an element to the set of reference layouts. We restrict our problem space further by enforcing non-overlapping of elements and focusing on flat layouts.

We tested two recently proposed methods to solve the layout completion problem, namely a graph neural network and a transformer model, as well as our own proposal which employs a sequence alignment algorithm and layout principles to generate features for a nearest neighbor search.

Our implementation of the graph neural network fails to produce high-quality results in the majority of cases and does not learn fine-grained patterns, but converges to few positions instead. At the same time, it produces a high number of overlapping predictions, making it unusable for a practical application.

The transformer model learns the positions as categorical tokens and employs a base grid that helps reduce the vocabulary size and helps generate well-aligned results. As a result, many predicted positions are valid, and many encoded patterns are correctly output in test cases. Overall, it is limited suitable due to drawbacks such as its inability to transfer patterns to different areas since a spatial understanding of a layout does not exist. Further, the lack of interpretability limits its applicability in the industrial

context.

Our nearest neighbor search finds insertion points via a sequence alignment algorithm and encodes layout principles in the feature generation to emphasize local similarity over global matches. Neighbors are then used to layout the new element based on the relations in the reference. The results show that layout patterns are returned with high accuracy, and it achieves overall satisfying alignment scores. Further, by considering the graph relations of layouts, it can adjust to shifted patterns in layouts. Its disadvantages are, however, that the sequence construction does not take columns or segments into account and no containment is supported. Further, its time and space complexity is dependant on the layout sizes and the size of the training data set, such that results require many seconds to be returned if more than a few hundred reference layouts are present. Finally, the sequence alignment algorithm is not tuned to the problem case which produces failures in certain edge cases. Nevertheless, its interpretability and high overall scores make it most suitable for practical usage for professional designers.

While we aimed for a rigorous, data-driven evaluation, we did not conduct a user study with the methods. Doing so would be beneficial to validating the layout assumptions and quality interpretations in this work. It can further reveal other limitations of the problem statement that are important for practical use.

Our analysis revealed that even today, classic methods of machine learning with principled methods for feature generation can be most suitable for practical applications in commercial environments where consistency of results and interpretability are important factors.

Future extensions of this research should address the limitations of the problem statement, i.e., supporting hierarchical elements and overlap to fully address the practical use cases. Further, while all methods can benefit from improvements, extending the kNN approach appears most promising by tuning the sequence alignment algorithm and improving the placement strategy. Finally, a user study should make the findings more robust.

With this work, we contribute to the understanding of the practical applicability of different methods for design assistance tools, and help companies to ensure consistency in their layouts.

Bibliography

[1] Steven Bradley. 2013. 4 Reasons Why You Should Design With A Grid. Retrieved 18.09.2020 from https://vanseodesign.com/web-design/why-grids/ 2.2

[2] Deng Cai, Shipeng Yu, Ji-Rong Wen, and Wei-Ying Ma. 2003. Vips: a vision-based page segmentation algorithm. (2003). 7

[3] Niranjan Damera-Venkata, José Bento, and Eamonn O'Brien-Strain. 2011. Probabilistic Document Model for Automated Document Composition. In *Proceedings of the 11th ACM Symposium on Document Engineering (DocEng '11)*. Association for Computing Machinery, New York, NY, USA, 3–12. 3.1, 7

[4] Niraj Dayama, Kashyap Todi, Taru Saarelainen, and Antti Oulasvirta. 2020. GRIDS: Interactive Layout Design with Integer Programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI '20)*. ACM. 1, 1.2, 2.2, 3.2

[5] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017. Rico: A Mobile App Dataset for Building Data-Driven Design Applications. In *Proceedings of the 30th Annual Symposium on User Interface Software and Technology (UIST '17)*. 6.1, 6.1, 6.1

[6] Krzysztof Gajos and Daniel S Weld. 2004. SUPPLE: automatically generating user interfaces. In *Proceedings of the 9th international conference on Intelligent user interfaces*. 93–100. 3.2

[7] Krzysztof Z. Gajos, Daniel S. Weld, and Jacob O. Wobbrock. 2010. Automatically generating personalized user interfaces with Supple. *Artificial Intelligence* 174, 12 (2010), 910 – 950.

[8] Krzysztof Z. Gajos, Jacob O. Wobbrock, and Daniel S. Weld. 2007. Automatically generating user interfaces adapted to users' motor and

vision capabilities. In *UIST '07: Proceedings of the 20th annual ACM symposium on User interface software and technology*. ACM Press, New York, NY, USA, 231–240. 3.2

[9] Inc. Google. 2020. Responsive layout grid. Retrieved 18.09.2020 from https://material.io/design/layout/responsive-layout-grid.html#columns-gutters-and-margins 1.2

[10] Edouard Grave, Piotr Bojanowski, Prakhar Gupta, Armand Joulin, and Tomas Mikolov. 2018. Learning Word Vectors for 157 Languages. In *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*. 4.5.3

[11] Kamal Gupta, Alessandro Achille, Justin Lazarow, Larry Davis, Vijay Mahadevan, and Abhinav Shrivastava. 2020. Layout Generation and Completion with Self-attention. (jun 2020). 1, 1.3, 3.3, 4.7, 4.4, 4.4.2, 4.4.3, 7

[12] Stephen M Hart and Liu Yi-Hsin. 1995. The application of integer linear programming to the implementation of a graphical user interface: a new rectangular packing problem. *Applied mathematical modelling* 19, 4 (1995), 244–254. 3.1

[13] Bruce Hillard, Jocelyn Amarego, and Tanya McGill. 2016. Optimising Visual Layout for Training and Learning Technologies. (2016). 4.2.1

[14] D. S. Hirschberg. 1975. A Linear Space Algorithm for Computing Maximal Common Subsequences. *Commun. ACM* 18, 6 (June 1975), 341–343. 4.5.2

[15] Forrest Huang, John F Canny, and Jeffrey Nichols. 2019. Swire: Sketch-based user interface retrieval. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–10. 3.3

[16] Justin Johnson, Agrim Gupta, and Li Fei-Fei. 2018. Image Generation from Scene Graphs. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. 1219–1228. 4.3.5

[17] Akash Abdu Jyothi, Thibaut Durand, Jiawei He, Leonid Sigal, and Greg Mori. 2019. LayoutVAE: Stochastic Scene Layout Generation from a Label Set. *CoRR* abs/1907.10719 (2019). 3.1

[18] Ranjitha Kumar, Jerry O Talton, Salman Ahmad, and Scott R Klemmer. 2011. Bricolage: example-based retargeting for web design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2197–2206. 3.2

[19] Markku Laine, Ai Nakajima, Niraj Dayama, and Antti Oulasvirta. 2020. Layout as a Service (LaaS): A Service Platform for Self-Optimizing Web Layouts. In *Web Engineering*, Maria Bielikova, Tommi Mikkonen, and Cesare Pautasso (Eds.). Springer International Publishing, Cham, 19–26. 3.2

[20] Chunggi Lee, Sanghoon Kim, Dongyun Han, Hongjun Yang, Youngwoo Park, Bum Chul Kwon, and Sungahn Ko. 2020. GUIComp: A GUI Design Assistant with Real-Time, Multi-Faceted Feedback. In *CHI '20: CHI Conference on Human Factors in Computing Systems, Honolulu, HI, USA, April 25-30, 2020*. ACM, 1–13. 3.3, 6.1

[21] Hsin-Ying Lee, Weilong Yang, Lu Jiang, Madison Le, Irfan Essa, Haifeng Gong, and Ming-Hsuan Yang. 2019. Neural Design Network: Graphic Layout Generation with Constraints. *ArXiv* (2019). arXiv:1912.09421 1, 1, 1.3, 3.1, 3.3, 4.2.2, 4.3, 4.4, 4.3.1, 4.3.5, 4.3.6, 5.2, 6.1, 6.1, 6.3, 7

[22] Luis A. Leiva. 2012. ACE: An Adaptive CSS Engine for Web Pages and Web-based Applications. In *Proceedings of the 21st international conference companion on World wide web (WWW)*. 3.2

[23] Luis A Leiva, Asutosh Hota, and Antti Oulasvirta. 2020. Enrico : A Dataset for Topic Modeling of Mobile UI Designs. In *22nd International Conference on Human-Computer Interaction with Mobile Devices and Services (MobileHCI '20 Extended Abstracts), October 5–8, 2020, Oldenburg, Germany (MobileHCI '20)*. ACM, 7. 6.1, 6.1

[24] Jianan Li, Jimei Yang, Aaron Hertzmann, Jianming Zhang, and Tingfa Xu. 2019. Layoutgan: Generating graphic layouts with wireframe discriminators. *7th International Conference on Learning Representations, ICLR 2019* (2019), 1–16. arXiv:1901.06767 3.1

[25] Yang Li, Julien Amelot, Xin Zhou, Samy Bengio, and Si Si. 2020. Auto Completion of User Interface Layout Design Using Transformer-Based Tree Decoders. *ArXiv* (2020), 1–11. 1, 1.3, 3.3, 4.4, 7

[26] Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101* (2017). 4.4.3

[27] Rafael Müller, Simon Kornblith, and Geoffrey E Hinton. 2019. When does label smoothing help?. In *Advances in Neural Information Processing Systems*. 4694–4703. 4.4.3

[28] Gene Myers. 1999. A Fast Bit-Vector Algorithm for Approximate String Matching Based on Dynamic Programming. *J. ACM* 46, 3 (May 1999), 395–415. 4.5.2

[29] Jakob Nielsen. 1994. Enhancing the Explanatory Power of Usability Heuristics. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '94)*. Association for Computing Machinery, New York, NY, USA, 152–158. 2.1

[30] Jakob Nielsen. 1994. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. 2.1

[31] Peter O'Donovan, Aseem Agarwala, and Aaron Hertzmann. 2014. Learning layouts for single-page graphic designs. *IEEE Transactions on Visualization and Computer Graphics* 20, 8 (2014), 1200–1213. 3.1

[32] Peter O'Donovan, Aseem Agarwala, and Aaron Hertzmann. 2015. Designscape: Design with interactive layout suggestions. In *Proceedings of the 33rd annual ACM conference on human factors in computing systems*. 1221–1224. 3.1

[33] Antti Oulasvirta, Niraj Ramesh Dayama, Morteza Shiripour, Maximilian John, and Andreas Karrenbauer. 2020. Combinatorial Optimization of Graphical User Interface Designs. *Proc. IEEE* 108, 3 (2020), 434–464. 2.2

[34] A. Ant Ozok and Gavriel Salvendy. 2000. Measuring consistency of web page design and its effects on performance and satisfaction. *Ergonomics* 43, 4 (2000), 443–460. 1

[35] Andreas Pfeiffer. 2018. *Creativity and technology in the age of AI*. Technical Report. Pfeiffer Consulting. Retrieved 18.09.2020 from http://www.pfeifferreport.com/essays/creativity-and-technology-in-the-age-of-ai/ 1

[36] Simo Santala. 2020-06-16. *Optimising User Interface Layouts for Design System Compliance*. Master's thesis. Aalto University. 1, 1.2, 3.2

[37] A. Sutcliffe. 2009. *Designing for User Engagement: Aesthetic and Attractive User Interfaces*. 1

[38] Amanda Swearngin, Chenglong Wang, Alannah Oleson, James Fogarty, and Amy J Ko. 2020. Scout: Rapid Exploration of Interface Layout Alternatives through High-Level Design Constraints. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–13. 3.2

[39] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2818–2826. 4.4.3

[40] Sou Tabata, Haruka Maeda, Keigo Hirokawa, and Kei Yokoyama. 2019. Diverse Layout Generation for Graphical Design Magazines. In *SIGGRAPH Asia 2019 Posters (SA '19)*. Association for Computing Machinery, New York, NY, USA, Article 21, 2 pages. 3.1

[41] Sou Tabata, Hiroki Yoshihara, Haruka Maeda, and Kei Yokoyama. 2019. Automatic Layout Generation for Graphical Design Magazines. In *ACM SIGGRAPH 2019 Posters (SIGGRAPH '19)*. Association for Computing Machinery, New York, NY, USA, Article 9, 2 pages. 3.1

[42] Jerry O. Talton, Lingfeng Yang, Ranjitha Kumar, Maxine Lim, Noah D. Goodman, and Radomír Měch. 2012. Learning design patterns with Bayesian grammar induction. *UIST'12 - Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology* (2012), 63–73. 3.1, 7

[43] Kashyap Todi, Daryl Weir, and Antti Oulasvirta. 2016. Sketchplore: Sketch and explore layout designs with an optimiser. *Conference on Human Factors in Computing Systems - Proceedings* 07-12-May- (2016), 3780–3783. 3.2

[44] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008. 4.4, 4.4.2

[45] Pengfei Xu, Hongbo Fu, Takeo Igarashi, and Chiew-Lan Tai. 2014. Global beautification of layouts with interactive ambiguity resolution. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*. 243–252. 3.2

[46] Xinru Zheng, Xiaotian Qiao, Ying Cao, and Rynson W.H. Lau. 2019. Content-aware generative modeling of graphic design layouts. *ACM Transactions on Graphics* 38, 4 (2019). 3.1

[47] Martin Šošić and Mile Šikić. 2017. Edlib: a C/C++ library for fast, exact sequence alignment using edit distance. *Bioinformatics* 33, 9 (01 2017), 1394–1395. 4.5.2, 5.2